Chapter 2

Writing Java Programs

Java Programs in General

- Building blocks of a Java program:
  - ***Classes.*** A class is a collection of related variables and/or methods (usually both). A Java program consists of one or more classes.
  - ***Methods.*** A method is a series of statements. Each class may contain any number of methods.
  - ***Statements.*** A statement is a single command. Each method may contain any number of statements.

Java Program Structure

- In the Java programming language:
  - A program is made up of one or more *classes*
  - A class contains one or more *methods*
  - A method contains program *statements*
- These terms will be explored in detail throughout the course
- A Java application always contains a method called `main`
- See `Lincoln.java`

```
//********************************************************************
//  Lincoln.java       Author: Lewis/Loftus
//
//  Demonstrates the basic structure of a Java application.
//********************************************************************

public class Lincoln
{
   //-----------------------------------------------------------------
   //  Prints a presidential quote.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      System.out.println ("A quote by Abraham Lincoln:");

      System.out.println ("Whatever you are, be a good one.");
   }
}
```

Java Program Structure

```
//  comments about the class
public class MyProgram
{



}
```

class header

class body

Comments can be placed almost anywhere

Java Program Structure

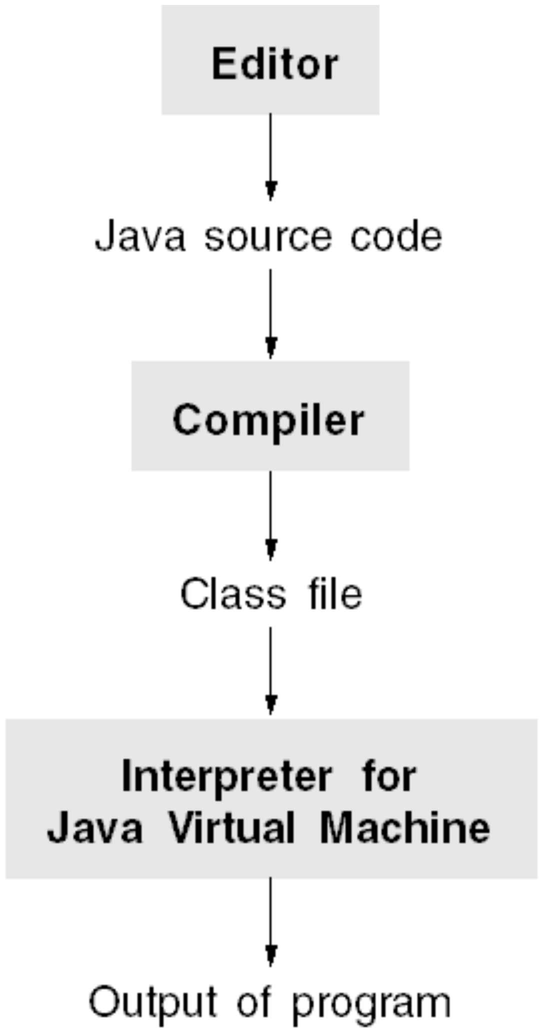
```
//  comments about the class
public class MyProgram
{

   //  comments about the method
   public static void main (String[] args)
   {

   }

}
```

method header

method body

Executing a Java Program

- Steps involved in executing a Java program:
  - Enter the program
  - Compile the program
  - Run the program
- With an integrated development environment (IDE), all three steps can be performed within the environment itself.


Editor
Java source code
Compiler
Class file
Interpreter for Java Virtual Machine
Output of program

Integrated Development Environments

- An ***integrated development environment*** (IDE) is an integrated collection of software tools for developing and testing programs.
- A typical IDE includes at least an editor, a compiler, and a debugger.
- A programmer can write a program, compile it, and execute it, all without leaving the IDE.

Write a Java Program without IDE

1. Install JDK,
2. Set the PATH variable
   - With administrator right, update PATH Variable
   - Use batch file, javacmd.bat
3. Any editor to enter a program, Notepad or WordPad.
   - save the file as a Text Document, Hello.java
4. Under DOS, compile the program, use the `javac`:

   ```
   javac Hello.java
   ```
5. Under DOS, run a program, use `java`:

```
java Hello
```

Comments

- They provide information that's useful for anyone who will need to read the program in the future.
- Typical uses of comments:
  - To document who wrote the program, when it was written, what changes have been made to it, and so on.
  - To describe the behavior or purpose of a particular part of the program, such as a variable or method.
  - To describe how a particular task was accomplished, which algorithms were used, or what tricks were employed to get the program to work.

Comments

- Comments should be included to explain the purpose of the program and describe processing steps
- They do not affect how a program works
- Java comments can take three forms:

```
// this comment runs to the end of the line

/*  this comment runs to the terminating
    symbol, even across line breaks        */

/** this is a javadoc comment   */
```

Types of Comments

- Single-line comments: `// Comment style 1`
- Multiline comments: `/* Comment style 2 */`
- "Doc" comments:

  `/** Comment style 3 */`
- Doc comments are designed to be extracted by a special program, `javadoc`.
- Forgetting to terminate a multiline comment may cause the compiler to ignore part of a program:

```
System.out.print("My ");    /* forgot to close this
comment...
System.out.print("cat ");
System.out.print("has ");   /* so it ends here */
System.out.println("fleas");
```

Single-line Comments

- Many programmers prefer // comments to /* … */ comments, for several reasons:
  - Ease of use
  - Safety
  - Program readability
  - Ability to "comment out" portions of a program

Tokens

- A Java compiler groups the characters in a program into ***tokens.***
- The compiler then puts the tokens into larger groups (such as statements, methods, and classes).
- Tokens in the `JavaRules` program:

```
public  class  JavaRules  {  public  static  void  main  (
String  [  ]  args  )  {  System  .  out  .  println  (
"Java rules!"  )  ;  }  }
```

Avoiding Problems with Tokens

- Always leave at least one space between tokens that would otherwise merge together:

```
publicclassJavaRules {
```

- Don't put part of a token on one line and the other part on the next line:

```
pub
lic class JavaRules {
```

Indentation

- Programmers use indentation to indicate nesting.
- An increase in the amount of indentation indicates an additional level of nesting.
- The `JavaRules` program consists of a statement nested inside a method nested inside a class:

```
public class JavaRules {

  public static void main(String[] args) {

    System.out.println("Java rules!");

  }

}
```

Brace Placement

- Put each left curly brace at the end of a line

```
public class JavaRules {
  public static void main(String[] args) {
    System.out.println("Java rules!");
  }
}
```

- Put left curly braces on separate lines

```
public class JavaRules
{
  public static void main(String[] args)
  {
    System.out.println("Java rules!");
  }
}
```

Brace Placement

- To avoid extra lines, the line containing the left curly brace can be combined with the following line:

```
public class JavaRules
{ public static void main(String[] args)
  { System.out.println("Java rules!");
  }
}
```

2.4 Using Variables

- In Java, every variable must be ***declared*** before use
- Declaring a variable means informing the compiler of the variable's name and its properties, including its ***type***.

```
   int i;   // Declares i to be an int variable
```

- Several variables can be declared at a time:

```
int i, j, k;
```

- ***variable declaration:***
  - The type of the variable
  - The name of the variable
  - A semicolon

Initializing Variables

- A variable is given a value by using =, the ***assignment operator:***

```
i = 0;
```

- Variables need to be initialized before the first use.
- Variables can be initialized at the time they're declared:

```
int i = 0;
```

- If several variables are declared at the same time, each variable can have its own initializer:

```
int i = 0, j, k = 1;
```

Changing the Value of a Variable

- The assignment operator can be used both to initialize a variable and to change the value of the variable later in the program:

```
i = 1;  // Value of i is now 1
…
i = 2;  // Value of i is now 2
```

Program: Printing a Lottery Number

Lottery.java

```
// Displays the winning lottery number

public class Lottery {
  public static void main(String[] args) {
    int winningNumber = 973;
    System.out.print("The winning number ");
    System.out.print("in today's lottery is ");
    System.out.println(winningNumber);
  }
}
```

Variable Types

- A partial list of Java types:

  `int` — An integer

  `double` — A floating-point number

  `boolean` — Either `true` or `false`

  `char` — A character

- Declarations of `double`, `boolean`, and `char` variables:

```
double x, y;
boolean b;
char ch;
```

Literals

- Represent a particular number or other value.
  - Examples of `int` literals: `0   297   30303`
  - Examples of `double` literals:

    `48.0   48.   4.8e1   4.8e+1   .48e2   480e-1`
- The only `boolean` literals are `true` and `false`.
- `char` literals are enclosed within single quotes:

  `'a'   'z'   'A'   'Z'   '0'   '9'   '%'   '.'   ' '`
- Literals are often used as initializers:

```
double x = 0.0, y = 1.0;
boolean b = true;
char ch = 'f';
```

2.6 Identifiers

- *Identifiers* are the "words" in a program
- A Java identifier can be made up of letters, digits, the underscore character (_), and the dollar sign
- Identifiers cannot begin with a digit,
- Java is *case sensitive*: `Total`, `total`, and `TOTAL` are different identifiers
- By convention, programmers use different case styles for different types of identifiers, such as
  - *title case* for class names - `Lincoln`
  - *upper case* for constants - `MAXIMUM`

Identifiers

- Sometimes the programmer chooses the identifier (such as `Lincoln`)
- Sometimes we are using another programmer's code, so we use the identifiers that he or she chose (such as `println`)
- Often we use special identifiers called *reserved words* that already have a predefined meaning in the language
- A reserved word cannot be used in any other way

Multiword Identifiers

- When an identifier consists of multiple words, it' s important to mark the boundaries between words.
- One way to break up long identifiers is to use underscores between words:

  ```
  last_index_of
  ```

- Another technique is to capitalize the first letter of each word after the first:

  ```
  lastIndexOf
  ```

  This technique is the one commonly used in Java.

Conventions

- A rule that we agree to follow, even though it's not required by the language, is said to be a ***convention.***
- A common Java convention is beginning a class name with an uppercase letter:

```
Color
FontMetrics
String
```

- Names of variables and methods, by convention, never start with an uppercase letter.

Keywords

- The following ***keywords*** can't be used as identifiers because Java has already given them a meaning:

`abstract`	`double`	`int`	`super`
`boolean`	`else`	`interface`	`switch`
`break`	`extends`	`long`	`synchronized`
`byte`	`final`	`native`	`this`
`case`	`finally`	`new`	`throw`
`catch`	`float`	`package`	`throws`
`char`	`for`	`private`	`transient`
`class`	`goto`	`protected`	`try`
`const`	`if`	`public`	`void`
`continue`	`implements`	`return`	`volatile`
`default`	`import`	`short`	`while`
`do`	`instanceof`	`static`	

- `null`, `true`, and `false` are also reserved.

Performing Calculations

- In general, the right side of an assignment can be an ***expression.***
- A literal is an expression, and so is a variable.
- More complicated expressions are built out of ***operators*** and ***operands.***
- In the expression `5 / 9`, the operands are `5` and `9`, and the operator is `/`.
- The operands in an expression can be variables, literals, or other expressions.

Operators

- Java's arithmetic operators: + – `*` `/` `%`

  `6 + 2` ⇒ 8

  `6 / 2` ⇒ 3

- Integer Division
  - If the result of dividing two integers has a fractional part, Java throws it away (we say that it ***truncates*** the result).

  `1 / 2` ⇒ 0

  `5 / 3` ⇒ 1

- ***unary*** arithmetic operators: require just one operand

  + Plus, +3

  – Minus, –3

- + and – are often used in conjunction with literals

`double` Operands

- `+`, `-`, `*`, and `/` accept `double` operands: Binary Operators

  `6.1 + 2.5` ⇒ 8.6
- `int` and `double` operands can be mixed:

  `6.1 / 2` ⇒ 3.05
- The `%` operator produces the remainder when the left operand is divided by the right operand:

  `13 % 3` ⇒ 1
- `%` is normally used with integer operands.

Round-Off Errors

- Calculations involving floating-point numbers can sometimes produce surprising results.
- If `d` is declared as follows, its value will be 0.09999999999999987 rather than 0.1:

```
double d = 1.2 - 1.1;
```

- ***Round-off errors*** such as this occur because some numbers (1.2 and 1.1, for example) can't be stored in `double` form with complete accuracy.

Operator Precedence

- What's the value of `6 + 2 * 3`?
  - `(6 + 2) * 3`, which yields 24?
  - `6 + (2 * 3)`, which yields 12?
- Operator precedence resolves issues such as this.
- `*`, `/`, and `%` take precedence over + and –.
- Examples:

```
5 + 2 / 2  ⇒  5 + (2 / 2)  ⇒  6
8 * 3 - 5  ⇒  (8 * 3) - 5  ⇒  19
6 - 1 * 7  ⇒  6 - (1 * 7)  ⇒  –1
9 / 4 + 6  ⇒  (9 / 4) + 6  ⇒  8
6 + 2 % 3  ⇒  6 + (2 % 3)  ⇒  8
```

Associativity

- Precedence rules are of no help when it comes to determining the value of `1 - 2 - 3`.
- Associativity rules come into play when precedence rules alone aren't enough.
- The binary `+`, `-`, `*`, `/`, and `%` operators are all left associative:

  `2 + 3 - 4` ⇒ `(2 + 3) - 4` ⇒ 1

  `2 * 3 / 4` ⇒ `(2 * 3) / 4` ⇒ 1

Parentheses in Expressions

- Parentheses can be used to override normal precedence and associativity rules.
- Parentheses in the expression `(6 + 2) * 3` force the addition to occur before the multiplication.
- It's often a good idea to use parentheses even when they're not strictly necessary:

```
(x * x) + (2 * x) - 1
```

- However, don't use too many parentheses:

```
((x) * (x)) + ((2) * (x)) - (1)
```

Assignment Operators

- = is used to save calculation result in a variable:

```
area = height * width;
```

- Assigning a `double` value to an `int` variable is not legal. Assigning an `int` value to a `double` variable is OK.
- = often uses the old value of a variable as part of the expression that computes the new value.

```
i = i + 1;
```

Compound Assignment Operators

- A partial list of compound assignment operators:

`+=`	Combines addition and assignment
`-=`	Combines subtraction and assignment
`*=`	Combines multiplication and assignment
`/=`	Combines division and assignment
`%=`	Combines remainder and assignment

```
i += 2;   // Same as i = i + 2;
i -= 2;   // Same as i = i - 2;
i *= 2;   // Same as i = i * 2;
i /= 2;   // Same as i = i / 2;
i %= 2;   // Same as i = i % 2;
```

Program: Converting from Fahrenheit to Celsius

FtoC.java

```
// Converts a Fahrenheit temperature to Celsius

public class FtoC {
  public static void main(String[] args) {
    double fahrenheit = 98.6;
    double celsius = (fahrenheit - 32.0) * (5.0 / 9.0);
    System.out.print("Celsius equivalent: ");
    System.out.println(celsius);
  }
}
```

Exercise: Write a Program: FtoC1

- Converting the following Fahrenheit to Celsius and print them out.

  86 32 100 201.2

- Output the average Celsius degree

- try: Increase each Fahrenheit by n=5 degree and do the conversion

Constants

- A ***constant*** is a value that doesn't change during the execution. Can be assigned to variables

```
double freezingPoint = 32.0;
double degreeRatio = 5.0 / 9.0;
```

- To prevent a constant from being changed, the word `final` can be added to its declaration:

```
final double freezingPoint = 32.0;
final double degreeRatio = 5.0 / 9.0;
```

- Constant names are often written in uppercase letters, with underscores to indicate boundaries between words:

```
final double FREEZING_POINT = 32.0;
final double DEGREE_RATIO = 5.0 / 9.0;
```

Adding Constants to the **`FtoC`** Program

`FtoC2.java`

```
// Converts a Fahrenheit temperature to Celsius

public class FtoC2 {
  public static void main(String[] args) {
    final double FREEZING_POINT = 32.0;
    final double DEGREE_RATIO = 5.0 / 9.0;
    double fahrenheit = 98.6;
    double celsius =
      (fahrenheit - FREEZING_POINT) * DEGREE_RATIO;
    System.out.print("Celsius equivalent: ");
    System.out.println(celsius);
  }
}
```

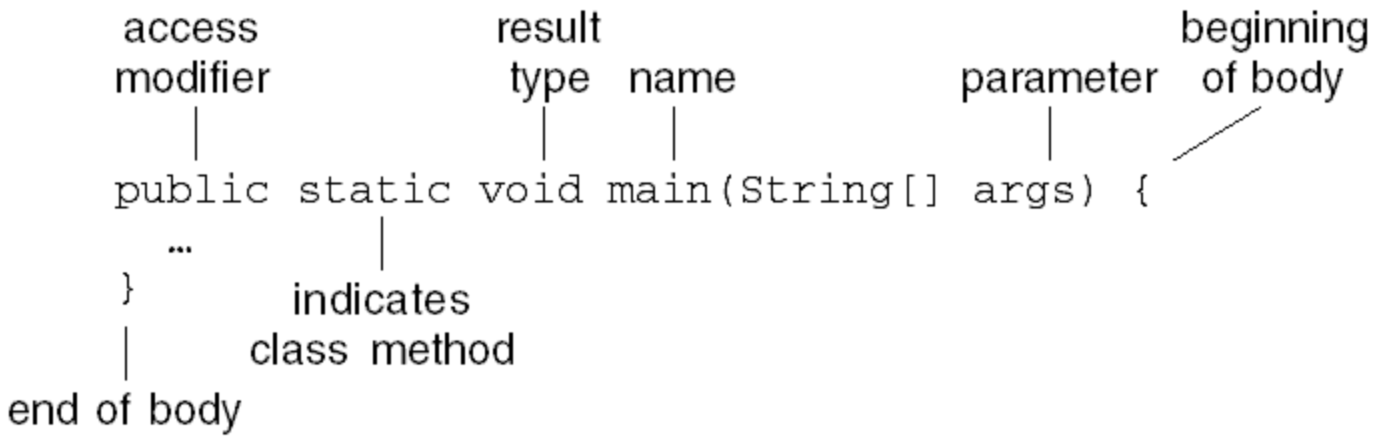
Methods

- A ***method*** is a series of statements that can be executed as a unit.
- A method does nothing until it is activated, or ***called.***
- To call a method, we write the name of the method, followed by a pair of parentheses.
- The method's ***arguments*** (if any) go inside the parentheses.
- A call of the `println` method:

```
System.out.println("Java rules!");
```

Declaring Class Methods

- Example of a class method declaration:



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

Parameters for Class Methods

- Java requires that `main`'s result type be `void` and that `main` have one parameter of type `String[]` (array of `String` objects).
- Other class methods may have any number of parameters, including none.
- If a method has more than one parameter, each parameter except the last must be followed by a comma.

Local Variables

- The body of any class or instance method may contain declarations of ***local variables.***
- Properties of local variables:
  - A local variable can be accessed only within the method that contains its declaration.
  - When a method returns, its local variables no longer exist, so their values are lost.
  - A method is not allowed to access the value stored in a local variable until the variable has been initialized.
  - A local variable can be declared `final` to indicate that its value doesn’t change after initialization.

The `return` Statement

- When a method has a result type other than `void`, a `return` statement must be used to specify what value the method returns.
- Form of the `return` statement:

  `return` *expression* `;`

- The expression is often just a literal or a variable:

```
return 0;
return n;
```

- Expressions containing operators are also allowed:

```
return x * x - 2 * x + 1;
```

Methods in the `Math` Class

- The `Math` class contains a number of methods for performing mathematical calculations.
- These methods are called by writing `Math.`*name*, where *name* is the name of the method.
- The methods in the `Math` class ***return*** a value when they have completed execution.

The `Math` Methods examples

- `pow` method raises a number to a power:

  `Math.pow(-2.0, 3.0)` ⇒ –8.0
- `sqrt` method computes the square root of a number:

  `Math.sqrt(4.0)` ⇒ 2.0
- `abs` method computes the absolute value of a number:

  `Math.abs(-2.0)` ⇒ 2.0
- `max` method finds the larger of two numbers:

  `Math.max(3.0, 5.5)` ⇒ 5.5
- The value returned by `abs`, `max`, and `min` depends on the type of the argument:
  - If the argument is an `int`, the methods return an `int`.
  - If the argument is a `double`, the methods return a `double`.

The `round` Method

- The `round` method rounds a `double` value to the nearest integer:

  `Math.round(4.1)` ⇒ 4
  `Math.round(4.5)` ⇒ 5
  `Math.round(-4.9)` ⇒ –5
  `Math.round(-5.5)` ⇒ –5

- `round` returns a `long` value rather than an `int` value.

Using the Result of a Method Call

- The value returned by a method can be saved in a variable for later use:

```
double y = Math.abs(x);
```

- Another option is to use the result returned by a method directly, without first saving it in a variable. For example, the statements

```
double y = Math.abs(x);
double z = Math.sqrt(y);
```

can be combined into a single statement:

```
double z = Math.sqrt(Math.abs(x));
```

Using the Result of a Method Call

- Values returned by methods can also be used as operands in expressions.
- Example (finding the roots of a quadratic equation):

```
double root1 =
   (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
double root2 =
   (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

- Because the square root of $b^2 - 4ac$ is used twice, it would be more efficient to save it in a variable:

```
double discriminant = Math.sqrt(b * b - 4 * a * c);
double root1 = (-b + discriminant) / (2 * a);
double root2 = (-b - discriminant) / (2 * a);
```

Using the Result of a Method Call

- The value returned by a method can be printed without first being saved in a variable:

```
System.out.println(Math.sqrt(2.0));
```

2.10 Input and Output

- Most programs require both input and output.
- ***Input*** is any information fed into the program from an outside source.
- ***Output*** is any data produced by the program and made available outside the program.

Displaying Output on the Screen

- Properties of `System.out.print` and `System.out.println`:
  - Can display any single value, regardless of type.
  - The argument can be any expression, including a variable, literal, or value returned by a method.
  - `println` always advances to the next line after displaying its argument; `print` does not.

Displaying a Blank Line

- One way to display a blank line is to leave the parentheses empty when calling `println`:

```
System.out.println("Hey Joe");
System.out.println();   // Write a blank line
```

- The other is to insert `\n` into a string that's being displayed by `print` or `println`:

```
System.out.println("A hop,\na skip,\n\nand a jump");
```

Each occurrence of `\n` causes the output to begin on a new line.

Escape Sequences

- The backslash character combines with the character after it to form an ***escape sequence:*** a combination of characters that represents a single character.
- The backslash character followed by `n` forms `\n`, the ***new-line character.***

Escape Sequences

- Another common escape sequence is `\"`, which represents `"` (double quote):

```
System.out.println("He yelled \"Stop!\" and we stopped.");
```

- In order to print a backslash character as part of a string, the string will need to contain two backslash characters:

```
System.out.println("APL\\360");
```

Printing Multiple Items

- The + operator can be used to combine multiple items into a single string for printing purposes:

```
System.out.println("Celsius equivalent: " + celsius);
```

- At least one of the two operands for the + operator must be a string.

Application Programming Interfaces

- The packages that come with Java belong to the Java ***Application Programming Interface*** (API).
- In general, an API consists of code that someone else has written but that we can use in our programs.
- Typically, an API allows an application programmer to access a lower level of software.
- In particular, an API often provides access to the capabilities of a particular operating system or windowing system.

Packages

- Java allows classes to be grouped into larger units known as ***packages.***
- Java comes with a large number of standard packages.
- Accessing the classes that belong to a package is done by using an ***import declaration:***

  `import` *package-name* `. * ;`

- Import declarations go at the beginning of a program. A program that needs `SimpleIO` or `Convert` would begin with the line

```
import jpb.*;
```

Program: Converting from Fahrenheit to Celsius (Revisited)

FtoC3.java

```
// Converts a Fahrenheit temperature entered by the user to
// Celsius

import java.util.Scanner;

public class FtoC3 {
  public static void main(String[] args) {
    final double FREEZING_POINT = 32.0;
    final double DEGREE_RATIO = 5.0 / 9.0;

    Scanner userInput=new Scanner(System.in);
    System.out.print(("Enter Fahrenheit temperature: ");
    double fahrenheit = userInput.nextDouble();
     double celsius =
      (fahrenheit - FREEZING_POINT) * DEGREE_RATIO;
    System.out.println("Celsius equivalent: " + celsius);
  }
}
```

Using Scanner for User Input

```
import java.util.Scanner;
public class GPAAverage2 {
  public static void main(String[] args) {
    Scanner userInput=new Scanner(System.in);
    System.out.print("Enter Course 1 score: ");
    double course1 = userInput.nextDouble();
    System.out.print("Enter Course 2 score: ");
    double course2 = userInput.nextDouble();

    double gpaAverage = (course1 + course2) / 2;
    System.out.println("\nGPA average: " + gpaAverage);
  }
}
```

Exercise: Write a Program: CourseAvg

- The `CourseAvg` program will calculate a class average, using the following percentages:

2 Programs	30%
2 Quizzes	10%
Test 1	15%
Test 2	15%
Final exam	30%

- The user will enter the grade for each program (0–100), the score on each quiz (0–10), and the grades on the two tests and the final (0–100).

Debugging

- ***Debugging*** is the process of finding bugs in a program and fixing them.
- Types of errors:
  - Compile-time errors
  - Run-time errors (called ***exceptions*** in Java)
  - Incorrect behavior

Fixing Compile-Time Errors

- Strategies for fixing compile-time errors:
  - Read error messages carefully. Example:

```
Buggy.java:8: Undefined variable: i
    System.out.println(i);
                          ^
Buggy.java:10: Variable j may not have been initialized
    System.out.println(j);
                          ^
```

  - Pay attention to line numbers.
  - Fix the first error.

Fixing Compile-Time Errors

– Don't trust the compiler (completely). The error isn't always on the line reported by the compiler. Also, the error reported by the compiler may not accurately indicate the nature of the error. Example:

```
System.out.print("Value of i: ")
System.out.println(i);
```

A semicolon is missing at the end of the first statement, but the compiler reports a different error:

```
Buggy.java:8: Invalid type expression.
    System.out.print("Value of i: ")
                    ^
Buggy.java:9: Invalid declaration.
    System.out.println(i);
                      ^
```

Fixing Run-Time Errors

- When a run-time error occurs, a message will be displayed on the screen. Example:

```
Exception in thread "main"
  java.lang.NumberFormatException: foo
      at java.lang.Integer.parseInt(Compiled Code)
      at java.lang.Integer.parseInt(Integer.java:458)
      at Buggy.main(Buggy.java:11)
```

- Once we know what the nature of the error is and where the error occurred, we can work backwards to determine what caused the error.

Fixing Behavioral Errors

- Errors of behavior are the hardest problems to fix, because the problem probably lies either in the original algorithm or in the translation of the algorithm into a Java program.
- Other than simply checking and rechecking the algorithm and the program, there are two approaches to locating the source of a behavioral problem, depending on whether a debugger is available.

Using a Debugger

- A ***debugger*** doesn't actually locate and fix bugs. Instead, it allows the programmer to see inside a program as it executes.
- Things to look for while debugging:
  - Order of statement execution
  - Values of variables
- Key features of a debugger:
  - Step
  - Breakpoint
  - Watch

Debugging Without a Debugger

- The JDK includes a debugger, named `jdb`.
- A debugger isn't always necessary, however.
- If a run-time error occurs in a Java program, the message displayed by the Java interpreter may be enough to identify the bug.
- Also, `System.out.println` can be used to print the values of variables for the purpose of debugging:

```
System.out.println("Value of a: " + a +
                   " Value of b: " + b);
```

Choosing Test Data

- Testing a program usually requires running it more than once, using different input each time.
- One strategy, known as boundary-value testing, involves entering input at the extremes of what the program considers to be legal.
- Boundary-value testing is both easy to do and surprisingly good at revealing bugs.