Interactive Input to Java Programs
Coding Style, Errors, Debugging
Indefinite Loops
Statement Flow Control

Input to Java Programs

- Interactive input makes most programs more interesting
  - Avoids hard coding of values
  - Program works with different input values without re-compiling
- Ways for a program to get input
  - Characters typed at keyboard
  - Mouse clicks and movements
  - Data read from file
  - Data read from a device, e.g., modem, ipod, etc.
- We will look at character input from keyboard
Input using Scanner

- Easy way for a program to obtain input
  - Does not use a graphical interface
  - Gets "console" input – text typed in a command prompt window
  - Versatile, e.g., input could be redirected to come from a file or from another program
  - Easy to parse input and perform conversions
- Uses `Scanner` class (Java 1.5 and later)
  - `Scanner` object is created
  - Various "next" methods are used to get input
  - Think of input as being stream of words to be handled
- Methods to get input
  - `next()` gets a String (a "word" marked by spaces)
  - `nextInt()` gets an integer (string of digits, converted)
  - `nextDouble()` gets a double (string of digits, ., etc., converted)
  - and more ... (see Scanner API documentation)

Input from Graphical Dialogs

- Generally, we will deal with "command line" programs, i.e., programs that run in a character based command prompt environment
  - Simple execution model – program starts from a shell, gets input from keyboard, sends output to command prompt
  - Command prompt in Windows, shell on Mac
- But even command line programs can pop up dialogs to accept typed input
  - Dialogs are a graphical interface
  - Our programs will not otherwise be graphical
- Graphical dialogs involve a lot of details
  - Creation of new windows
  - Painting of bitmapped images on screen
  - Responding to keyboard input and mouse clicks
- Fortunately, the Java library provides methods to hide all these details and make it easy to use a dialog
Input Dialog

- An input dialog has several attributes:
  - Title, Icon, Prompt, Text box, buttons

- What form of input can we expect from an input dialog?
  - We type in characters from the keyboard, so the input is a bunch of characters
  - Thus, we get a String from the dialog
  - If the cancel button was clicked, then we get nothing, which is different from not typing anything and clicking OK

- The input dialog method returns a String
  - The String is the characters that were typed
  - null is returned if cancel was chosen

How to code an Input Dialog

```java
String response = JOptionPane.showInputDialog( null, 
  "Enter a temperature:",
  "Temperature to convert",
  JOptionPane.QUESTION_MESSAGE);
```

InputDialog.java
Converting Strings

- Sometimes we want to interpret the input as a numerical value instead of a string
  - But the user can only type characters (although some characters may be digits)
  - And the input dialog can only return a String
- Method to convert a String to an integer
  - Method `Integer.parseInt`
  - Takes a String, returns an integer
  - Analyzes characters, and returns corresponding numerical value
  - Not at all the same as casting
- Method to convert a String to a double
  - Method `Double.parseDouble`

Case Study: Payment Calculator

- Problem: compute monthly payments for a loan
  - Useful formula for payment:
    \[ L \times \left( r + \frac{r}{(1 + r)^n - 1} \right) \]
  - \( L \) is original loan amount, \( n \) is number of payments, \( r \) is interest rate for payment period
- Strategy for program to solve problem
  - Use `Scanner` to get loan amount, annual interest, length in years
  - Calculate number of monthly payments (12 months per year)
  - Calculate monthly interest rate
  - Use formula to calculate monthly payment
  - Output result
  - Use `Math.pow` to handle exponent in formula
Formatting Output

- System.out.print automatically formats integers and doubles
  - Precision of doubles may not be what we want
  - Format objects can be created for locale specific output and other customizations
  - This model generalizes to other output types (e.g., dates, money)
- New in Java 5: System.out.printf provides simple formatting
  - Uses a control string containing %d, %f, etc. with precision specifications
  - Additional parameter expressions must be provided to match all the % controls
  - Will generate exceptions if missing expressions or mis-matched types
  - Can also use to pad with leading zeroes, blanks
- LoanPayment2.java

Coding Style

- How a program looks is important
  - Easier to read and understand
  - Easier to spot errors
  - Reflects organized design and thinking
- Include appropriate comments
- Use indentation to show structure
  - Blocks and nested blocks
  - Use white space to increase readability
  - Align closing brace with block beginning
- Naming conventions
  - Use meaningful class, variable, and method names
  - Capitalize class names
- Designing and coding a program is a creative and artistic activity
Programming Errors

- Compiler Errors
  - Error messages from the compiler are syntax or semantic errors
  - The compilation fails, and no executable program is produced
  - Examples of syntax errors:
    - missing semi-colons, braces, parentheses
    - improper use of keywords, illegal use of operators
  - Examples of semantic errors
    - Variable used but not defined
    - Assignment of double to int without cast
    - Missing import causes undefined class
  - One error (e.g., a missing brace) may lead to many others
    - Sometimes it's best to fix first few, then compile again

- Runtime Errors
  - Program compiles successfully, but throws exception when run
  - Some illegal condition has occurred that means the program cannot continue to safely execute
  - Examples:
    - Division between two variables where denominator value is zero
    - Attempt to use parseInt on a String that is not all digits
    - No static main method in class being executed
  - Fix runtime errors by coding in logic to avoid those situations

- Logic Errors
  - Program compiles and runs, but produces incorrect results
  - Code is okay, but does not do what you intended
  - Fix by changing code to correspond to what you want to do
Debugging

IDEs (Integrated Development Environments) provide sophisticated debugging
- Display values of variables
- Trace execution
- Set breakpoints to stop at critical places

For small programs
- Use print statements to display crucial values
- Use print statements for tracing (e.g., does program get here?)
- Comment out blocks of code to isolate problems
- Edit, compile, and test programs incrementally
  - Start with small amount of working code and add to it

Conditional Evaluation

All evaluation is left-to-right, including logical operators && and ||
But these operators also have the property of "short circuit" evaluation
- For a logical AND, if the first term is false, then there is no reason to evaluate the second term, since the expression would still be false
- For a logical OR, if the first term is true, then there is no reason to evaluate the second term, since the expression would still be true

This is called conditional evaluation
- More efficient – irrelevant code is not executed
- Makes it easy to do defensive checks
  - Example: `if (n == 0 || m%n != 0) // m not divisible by n`
- Or if there are side effects
  - Example: `if (i < s.length() && s.charAt(i++) == ' ')`
Loops

- Many problem solutions involve the repetition of a task for an indefinite number of times
  - Taking turns in a game
  - Finding an answer by approximation – make repeated "better" guesses
  - Put a list of numbers or names in order
- Programming languages have looping control flow constructs
  - Fundamental model is to repeatedly test and perform task, stopping when test condition is no longer true
- Java has three forms of loops
  - while, do-while, for
  - Have already seen how to use a for loop for fixed repetitions

While Loop

- The while loop has a test condition and a body
  - Test condition is a boolean expression
  - Body is a single statement, or a block of statements
- If the condition is true, the body is executed
  - Then the condition is checked again
- Syntax of while statement
  - `while (condition) {
    statements;
  }

  `condition` is a boolean expression
  `statements` is required if more than one statement
  `while` is the keyword
Flow chart of while loop

```java
howmany = 3; start = 5;
number = 7; sum = 18; count = 4;

while (count <= howmany) {
    ++number;
    sum += number;
    ++count;
}

System.out.println("Sum from " + start + " to " + number + " is " + sum);
```

Do-While Loop

- Similar to a while loop, but the condition is checked after the body of statements
  - Useful when we want to execute at least once
  - If the condition is true, the body is executed again
  - Then the condition is checked again

Syntax of do-while statement

```java
do {
    statements;
} while (condition);
```
Flow chart of do-while loop

```
target = 3;
guess = 3;

do {
    System.out.print("Guess ");
guess = scan.nextInt();
} while (guess != target);

System.out.println("Right!");
```

Guess.java

For Loop

- Loops are often used to execute a block of statements a fixed number of times
  - We can do this with a while loop by initializing a counter and using a condition to check the counter, and incrementing the counter as the loop progresses
  - Have seen how to do fixed iterations with a for loop
- The for loop can be used for indefinite loops
  - Can be used wherever a while loop is used
  - A for loop helps distinguish between the mechanics of the looping and the work that is to be done repeatedly
The general for loop has four parts:
- Initialization, which is only done once
- Condition, which is checked for each iteration
- Body which is executed if the condition is true
- Iterator (usually an increment), which is executed after the body, each time the body is executed

After body and iterator are executed, condition is checked again.

Syntax of for statement:
```
for (initialize; condition; iterate) {
    statements;
}
```

- expression evaluated once, may be a declaration
- keyword `for`
- `initialize`;
- `condition`;
- `iterate`;
- `statements;`
- parentheses required
- "increment" done after the statements to get to next iteration
- semicolons required

Each of the control parts of a for loop may be omitted:
- If the initialization part is left out, there is no initial code to execute
  - Variables have values from previous statements
- If the condition is left out, the loop test is always true
  - An "infinite" loop, so there must be other code to terminate
- If the increment is left out, there is no increment to execute
  - There should be other code to advance the loop
  - In this case, we probably would use a while loop instead
- Multiple variables can be initialized and incremented
  - Can be useful, but may make loop hard to understand
More on Loops

- If condition never becomes false, the loop will execute "forever"
  - An infinite loop will continue until some program fault is reached
  - The program may need to be externally terminated
  - Make sure when coding a loop that you provide logic to advance the condition toward termination
  - Logic may be decrement, dividing by two, multiplying by 3, ...
    - Logic must guarantee that condition is eventually false
- Loops are a type of statement
  - A loop can be selected by an if or if-else
  - Loop bodies may contain if, if-else, or other loop statements
  - So loops may be nested

Break and Continue

- The **break** statement causes the early termination of a loop
  - Control flows immediately to the statement which is right after the loop
  - Syntax is: **break**;
- The **continue** statement jumps to the next iteration of the loop
  - Control flows immediately **back** to the loop control, skipping the rest of the statements of the loop
  - In a for loop, control skips to the increment, then to the condition
  - Syntax is: **continue**;
Flow of break and continue

```java
int sum = 0;
for (int count = 0; count < 10; ++count) {
    System.out.print("Enter a number: ");
    if (!input.hasNext()) {
        System.out.println("End of input");
        break;
    } else if (!input.hasNextInt()) {
        String s = input.next();
        System.out.println("not a number");
        continue;
    }
    int number = input.nextInt();
    sum += number * number;
}
System.out.println("sum is "+sum);
```

SquareSum.java

Switch Statements

- If choice among multiple alternatives is based on an integral expression, a `switch` construct can be used
- Controlling value could be byte, int, char
- Choices are listed explicitly (no comparisons like < or >)
- Keyword `default` is used for value not explicitly matched
- Switch acts like a table transferring control to the matching statement(s)

Syntax of switch statement

```java
switch (expression) {
    case value1:
        statements;
        break;
    case value2:
        statements;
        break;
    . . .
    default:
        statements;
        break;
}
```
Flow control of switch statement

```
v = 0.0;
switch(type) {
  case 0:
    v = len * w * h;
    break;
  case 1:
    v = 0.5 * len * w * h;
    break;
  case 2:
    v = (1.0/3) * len * w * h;
    break;
  default:
    System.err.println ...
    break;
}
System.out.println("Volume is "+v);
```

Flow control of switch statement

```
switch transfers into case list
v = len * w * h;
break;

normal sequential execution

v = 0.5 * len * w * h;
break;

break transfers out of case list

v = (1.0/3) * len * w * h;
break;

default:
  System.err.println ...

System.out.println("Volume is "+v);
```

Java Strings

- We have been using the Java `String` type in programs
  - We initialize Strings with literals in double quotes, e.g.,
    ```
    String msg = "Hello, world";
    ```
  - We can perform some operations, e.g.,
    ```
    msg = msg + "!";
    ```
  - Numbers can convert to Strings
    ```
    msg = msg + " guest number " + n;
    ```
  - Strings are objects, not primitive types
    - Strings have methods, e.g., `msg.length()`
    - Strings can be created with new, e.g.,
      ```
      String msg = new String("Hello, world");
      ```
    - Initialization from literal is just shorthand for new
    - Various constructors for Strings to create Strings from literals, other Strings, arrays of characters, etc.
    - Void constructor for String creates empty string, i.e., ""
String Methods

- Return character at given index (offset from beginning)
  
  ```java
  String s = "Go Ducks";
  s.charAt(1) is 'o'
  s.charAt(4) is 'u'
  s.length() is 8
  ```

- Form string by pasting two together
  
  ```java
  s.concat("!!!") is "Go Ducks!!!"
  This is a new String, s is unchanged (like s+"!!!")
  ```

- Extract a substring
  
  ```java
  s.substring(3,8) is "Ducks" (another new String)
  ```

- Produce capitalization (also can do lower case)
  
  ```java
  s.toUpperCase() is "GO DUCKS!!!" (another new String)
  ```

- Many more methods String API page
  
  - But none of them change the object