Intermediate Programming with Java

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

Concepts

- You should already have some programming background:
  - Java, Python, C, C++, VB, etc

Concepts

- You should already have some programming background:
  - Java, Python, C, C++, VB, etc
- Familiar with the following concepts:
  - Programming structures and syntax
    - How to write a program (source code)
    - How to run/execute a program

Concepts

- You should already have some programming background:
  - Java, Python, C, C++, VB, etc
- Familiar with the following concepts:
  - Programming structures and syntax
    - How to write a program (source code)
    - How to run/execute a program
  - Types and Expressions
    - Integer, floating-point, etc
    - Operators and precedence

Concepts

- You should already have some programming background:
  - Java, Python, C, C++, VB, etc
- Familiar with the following concepts:
  - Programming structures and syntax
    - How to write a program (source code)
    - How to run/execute a program
  - Types and Expressions
    - Integer, floating-point, etc
    - Operators and precedence
  - Control statements and decisions
    - Boolean expressions (true/false)
    - `if` and `switch` statements
    - Loops (*for* and *while*)

Concepts

- Familiar with the following concepts:
  - Methods or functions, parameters, and return values
    - Implementation
    - Calling with parameters (arguments)
    - Get a return value

Concepts

- Familiar with the following concepts:
  - Methods or functions, parameters, and return values
    - Implementation
    - Calling with parameters (arguments)
    - Get a return value
  - Arrays and their uses
    - One-Dimensional array
    - How to access elements

Concepts

- Familiar with the following concepts:
  - Methods or functions, parameters, and return values
    - Implementation
    - Calling with parameters (arguments)
    - Get a return value
  - Arrays and their uses
    - One-Dimensional array
    - How to access elements
- **If you do not have these background**, CS0007 or CS0008.

Goals

1. To quickly cover the basics of the Java programming
   - From implementation point-of-view, not concepts
     - Basic syntax, and how to compile and run
   - If you already know how to write program, it will be straightforward
     - Learn and understand Java's syntax
   - Foundations of Object-Oriented Programming
     - Classes and Objects and how to implement and use them
   - Chapters 1 to 5 of our textbook

Goals

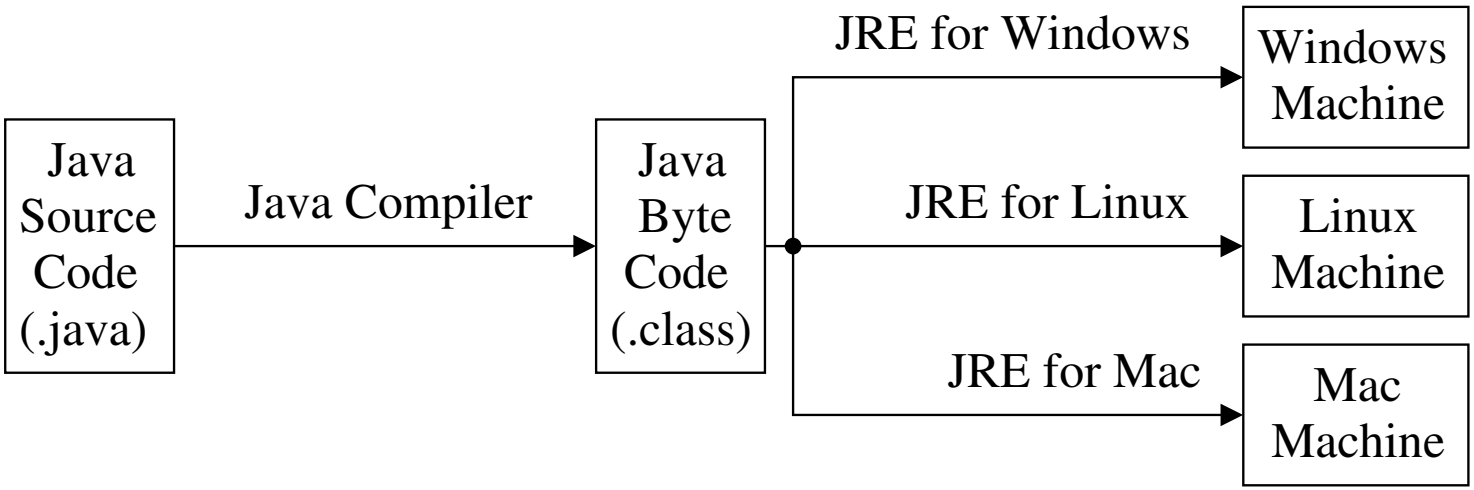
2. To learn the Principles of Object-Oriented Programming
   - Learn from Java point-of-view of OOP
   - Objects, methods, and instance variables
     - References and their implications
   - Creating new classes
     - syntax and logic required
   - Inheritance and composition
     - Building new classes from old classes
   - Polymorphism and dynamic binding
     - Accessing different objects in a uniform way
   - Chapters 6, 8 - 10 of our textbook
     - We will focus a lot on these topics
   - Why OOP is good an how to do it in Java
     - Can be applied with other OOP languages C++, Object-C, C#, Python, etc

Goals

3. To learn additional useful programming techniques in Java
   - Array and linked-list use and algorithms (sorting, searching) (Chapter 7)
   - Reading and writing files (Chapters 4 and 11)
   - Exception Handling (Chapter 11)
   - Graphical User Interface and Applications (Chapters 12 – 14)
   - Introduction to recursion (Chapter 15)

Why Java?

- Java is an **interpreted**, **platform independent**
  - A Java source code (`.java`) is compiled into an intermediate code (byte code)
  - A byte code does not run directly on a machine
    - Another software (interpreter) is the one that execute a byte code
    - This interpreter is called Java Runtime Environment (JRE)
    - Imagine that JRE is a **Virtual** machine



Why Java?

- Java is an **interpreted**, **platform independent**
  - Benefits
    - Safety features and run-time checks can be built into the language
    - Same source code can be run in multiple platforms
    - Supported JRE must be installed

Why Java?

- Java is an **interpreted**, **platform independent**
  - Benefits
    - Safety features and run-time checks can be built into the language
    - Same source code can be run in multiple platforms
    - Supported JRE must be installed
  - Drawback:
    - Slow compare to C or C++

Why Java?

- Java is an **interpreted**, **platform independent**
  - Benefits
    - Safety features and run-time checks can be built into the language
    - Same source code can be run in multiple platforms
    - Supported JRE must be installed
  - Drawback:
    - Slow compare to C or C++
- Java is an **Object-Oriented** language
  - We will discuss this later

Compile and Run a Java Program

- First you need to install **Java Development Kit** (JRE included)

  ▸ Oracle's Java SE Edition Download

- A source code (e.g., `IAmGroot.java`) is a plain text file
- To compile on a console (`cmd` or terminal):

```
javac IAmGroot.java
```

  if all go well, the byte code `IAmGroot.class` will be created.
  - More class files may be created (talk about this later)
- To run/interpret on a console (`cmd` or terminal):

```
java IAmGroot
```

- Lab 1 wiill introduce you to JDK

I Am Groot...

- Let's look at the source code of `IAmGroot.java`

```
public class IAmGroot
{
    public static void main(String[] args)
    {
        System.out.println("I am Groot...");
    }
}
```

- Output

```
I am Groot...
```

Integrated Development Environment (IDE)

- Most developers use Integrated Development Environment (IDE)
  - Two most popular are:
    - Eclippse
    - Netbean
  - Integrates editing, compiling, debugging, and running in one
    - Syntax suggestion
    - Breakpoint
    - Monitor variable values
  - In the end, if gives you source files (`.java`) and class files `.class`
    - Source files from IDE can be compile by `javac`
    - Class files from IDE can be executed by `java`
  - Feel free to use IDE
- Check `ex1.java` — **Carefully read the comments!**

Necessary Abilities from Java

- The are some basic necessity from Java
  - Able to take an input as well as produce an output (I/O)
    - Input: keyboard, mouse, touch, files, etc
    - Output: screen, files, etc

Necessary Abilities from Java

- The are some basic necessity from Java
  - Able to take an input as well as produce an output (I/O)
    - Input: keyboard, mouse, touch, files, etc
    - Output: screen, files, etc
  - Able to create and name variables and constants to store data
    - Identifiers and variable declarations

Necessary Abilities from Java

- The are some basic necessity from Java
  - Able to take an input as well as produce an output (I/O)
    - Input: keyboard, mouse, touch, files, etc
    - Output: screen, files, etc
  - Able to create and name variables and constants to store data
    - Identifiers and variable declarations
  - Able to manipulate and operate on data
    - Statements
    - Expressions

Necessary Abilities from Java

- The are some basic necessity from Java
  - Able to take an input as well as produce an output (I/O)
    - Input: keyboard, mouse, touch, files, etc
    - Output: screen, files, etc
  - Able to create and name variables and constants to store data
    - Identifiers and variable declarations
  - Able to manipulate and operate on data
    - Statements
    - Expressions
  - Able to make decisions
    - Control Structures

Input/Output

- Output (for now)
  - Java has a predefined object called `System.out`
  - This object has the ability to output data to the **standard output stream** (console screen)
    - Simply use its methods (`print()`, `println()`, `printf()`, ...)
  - Need to supply information (parameters/arguments)

```
System.out.println("I am Groot...");
```

  - We can output strings, values of variables, and expressions

Lexical Elements

- Lexical elements are groups of characters used in program code
  - They form all of the program code:
    - Keywords, identifiers, literals, delimiters
- **Keywords**:
  - Have predefined meaning in the language
  - We cannot redefine or used in another way
  - Examples: `if`, `else`, `class`, `int`, ...
  - See page 10 of our textbook for a complete list of Java keywords

Lexical Elements

- **Predefine Identifiers**:
  - Identifiers as part of some predefine classes/packages
    - **Class Names**: `System`, `JFrame`, `ArrayList`
    - **Method Names**: `println`, `close`, `add`
    - **Constant Names**: `PI`, `E`
  - We can use within their defined contexts
  - Can be redefined but under some restrictions
  - Java has tons of these predefine identifiers
    - Java comes with a huge collection of predefine classes

Lexical Elements

- **Other Identifiers**:
  - Defined by programmer
  - Used to represent names of variables, classes, methods, etc
  - Cannot be a **keyword**
  - Can be the same as a predefined identifier
    - Again, under some restrictions
  - An identifier must:
    1. begin with a letter (`a` – `z` or `A` – `Z`), or the underscore (`_`)
    2. followed by zero or more letters, digits (`0` – `9`), the underscore, or the dollar sign (`$`) characters

Identifiers and Naming Conventions

- Identifiers are **case-sensitive**
  - `data` and `Data` are two different identifiers
- **Naming Conventions**:
  - **Class Names**: start with an upper case and start each word with an upper case
    - Examples: `ArrayList`, `StringBuilder`, `IAmGroot`
  - **Method and Variable Names**: start with a lower case letter and start each word with an upper case letter
    - Examples: `indexOf`, `isEmpty`, `compareTo`

Literals

- Literals are values that are hard-coded into a program
  - Do not have to be numbers
- Different types have different rules for literal values
  - **Integer**: optional +/− symbol followed by one or more digits (0 – 9)
    - Examples: `12`, `-345`, `+9`
  - **String**: a sequence of characters (letters, digits, space, symbols) contained within double quotes
    - Examples: `"Hello"`, `"I am Groot..."`
  - **Character**: a single character (letter, digit, space, symbol) contained within single quotes
    - Examples: `'5'`, `'a'`, `' '`

Lexical Elements

- Building blocks of Java programs
  - **Keywords**:
    - Restricted to their predefined use
  - **Predefine Identifiers**:
    - Predefined by Java but can be redefined
  - **Programmer Defined Identifiers**:
    - Created by a programmer (variable names, class names, method names, etc)
  - **Literals**:
    - Hard-coded values into a program

Statements

- A **Statement** is a unit of declaration or execution
- Executing a program is the same as executing a sequence of statements
- Every statement in Java must be ended by a **semicolon** (;)
- **Variable Declaration Statements**:

```
int count;
float price;
```

- **Assignment Statements**:

```
count = 2;
price = 12.34;
```

- **Method Calls**:

```
System.out.println("I am Groot...");
```

- Check `ex2.java` for more examples

Variables

- A **variable** is a memory location that is associated with an identifier
  - Given an empty notebook with page numbers
    - Write a name (e.g., `myData`) and a page number (e.g., 37) in a piece of paper
    - Page 37 can be referred by the name `myData`
    - You can write, erase, or modify data on the page 37
    - You can also refer to data in another page (e.g., another name and another page number on the page 37)
  - Variables are generally used to store data (e.g., numbers, characters, objects, etc)
  - Data can be changed throughout the executing of a program
  - A **type** or a **class** must be specified for each variable

Variables

- The type of a variable specifies its properties
  - Data it can store (e.g., number, character, etc)
  - Operation that can be performed on it (e.g., convert to string, etc)
  - Compilation error will occur if we try to store **incorrect data type**:

```
int x = "Hello";
```

Error Message:

```
error: incompatible types: String cannot be converted to int
        int x = "Hello";
                ^
```

`x` can be used to store an integer but `"Hello"` is a string

Variables

- The compilation error also occurs if a precision is lost

```
int x = 2.7;
```

Error message:

```
error: incompatible types: possible lossy conversion from
       double to int
        int x = 2.7;
                ^
```

  - `x` is not precise enough to store 2.7
    - `x` can store either 2 or 3

- Numeric type in Java from lowest precision to highest

$$\texttt{byte} < \texttt{int} < \texttt{long} < \texttt{float} < \texttt{double}$$

- It is okay the other way around

```
double y = 100;
```

Floating-Point Variables

- In Java, a floating-point number is `double` (by default)

```
float pi = 3.14;
```

Error message:

```
error: incompatible types: possible lossy conversion from
       double to float
        float pi = 3.14;
                   ^
```

- You must **explicitly** cast to less precise

```
float pi = (float) 3.14;
```

Exercise

- Compilation Error?
  - `int i = 5;`

    Okay
  - `int j = 4.5;`

    Not okay
  - `float x = 3.5;`

    Not okay
  - `float y = (float) 9.2;`

    Okay
  - `double z = 100;`

    Okay
  - `i = z;`

    Not okay as well as `y = z;` but `z = i;` is okay
  - `j = (long) y;`

    Not okay but `j = (byte) y;` is okay

Primitive Type Variables

- There are two main categories of variables
  - **Primitive Types**:
    - Simple types: Value stored directly in the memory location
    - Examples:

```
int x = 100;
double pi = 3.14;
```

    - In Java, there are 8 primitive types:

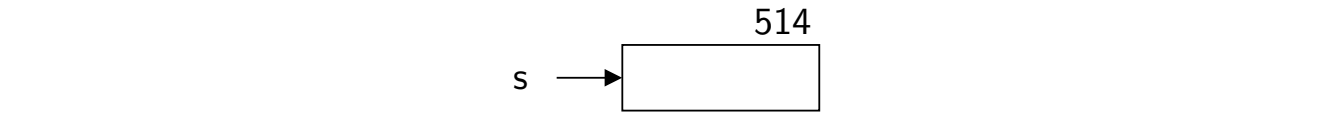
      `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`

- Check `ex3.java` for more details on the primitive numeric types

Reference Type Variables

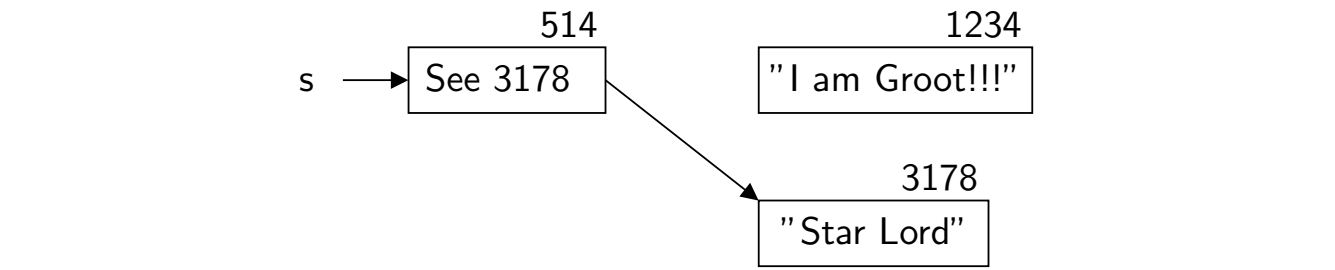
- There are two main categories of variables
  - **Reference Types** (class types):
    - Values are **reference** to **object** that are stored elsewhere in memory
    - Example:

```
String s;
```




```
s = new String("I am Groot!!!");
```


```
s = new String("Star Lord");
```



    - Objects of different types have different capabilities

Rules for Declaration and Use of Variables

- All variables must be declared before they can be used
- Use without declaration:

```
x = 20;
```

Error message:

```
error: cannot find symbol
        x = 20;
        ^
```

- Example: **Declare and then use**

```
int x;
x = 20;
```

- Example: **Initialize during declaration**

```
int x = 20;
```

Rules for Declaration and Use of Variables

- Multiple variables with the same type can be declared in a single declaration statement:

```
int i, j, result = 0, count = 0, max = 10;
```

  which is equivalent to

```
int i;
int j;
int result = 0;
int count = 0;
int max = 10;
```

- Multiple declaration statements are required for multiple variables with different types:

```
int counter = 0, max = 10;
double average = 0.0;
```

Operators and Expressions

- Numeric Operators in Java:

$$+, -, *, /, \text{ and } \%$$

- Pretty much the same as in other programming language
- Notes:
  - If both operands are integers / gives an integer result

```
int x = 5 / 2;          // x is 2   (integer division)
double y = 5 / 2;       // y is 2.0 (integer division)
double z = 5.0 / 2;     // z is 2.5 (floating-point division)
```

  - The modulo operator (%) can be used with both integers and floating-point

```
int x = 5 % 2;          // x is 1   (integer modulation)
double y = 5 % 2;       // y is 1.0 (integer modulation)
double z = 5.1 % 2.0    // z is 1.1 (floating-point modulation)
```

Precedence and Associativity

- **Precedence** indicates the order in which operators are applied in an expression
  - `*`, `/`, and `%` have the same precedence
  - `+` and `-` have the same precedence but lower than `*`, `/`, and `%`
- **Associativity** indicates the order in which operands are accessed given operations of **the same precedence**
  - `+`, `-`, `*`, `/`, and `%` are left-to-right associativity
- Examples:
  - `a + b * c` $\rightsquigarrow$ `a + (b * c)`
  - `a * b / c` $\rightsquigarrow$ `(a * b) / c`
  - `a + b * c - d` $\rightsquigarrow$ `(a + (b * c)) - d`
- Use parentheses to eliminate ambiguity

Operators (Shorthand Notations)

- Allow us to do operations with less typing
- Examples:
  - `x = x + 1;` ⇝ `x++;`
  - `y = y - 7;` ⇝ `y -= 7;`
- +=, −=, *=, /=, %= are available
- Be careful:
  - `x *= y + 5;` ⇝ `x = x * y + 5;` ⇝ `x = (x * y) + 5;`
  - `x *= (y + 5);` ⇝ `x = x * (y + 5);`
- **Prefix** vs **Postfix** of unary operators
  - Both `x++` and `++x` perform `x = x + 1;`

```
public class TestUnary {
    public static void main(String[] args) {
        int x = 5, y = 5;
        System.out.println(x++);  // output 5
        System.out.println(x);    // output 6
        System.out.println(++y);  // output 6
        System.out.println(y);    // output 6
    }
}
```

Input

- Java has a predefined object called `System.in`
  - Similar to `System.out` discussed earlier
    - Output to **output stream** (console)
  - Allows data to be input from the **standard input stream**
- Generally, this object is used to read data from the keyboard
- Previously, input is difficult to implement
  - The new Java provides the `Scanner` class to help with this

The Scanner class

- The `Scanner` class reads data from a standard input stream (e.g., keyboard) and parses it into tokens based on a **delimiter**
  - A **delimiter** is a character or set of characters that distinguish one **token** from another
  - A **token** is all of the characters between delimiters
  - By default, the `Scanner` class uses white space characters (space, tab, and newline) as the delimiters
- The tokens can be read in either as
  - **String**: `next()`, `nextLine()`, or
  - **Primitive Types**: `nextInt()`, `nextFloat()`, or `nextDouble()`

Problem with the Scanner Class

- If read as primitive types, an error will occur if the actual token does not match what you are trying to read
  - For example, use `nextInt()` to read an integer but a user types `Hello`

```
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:864)
at java.util.Scanner.next(Scanner.java:1485)
at java.util.Scanner.nextInt(Scanner.java:2117)
at java.util.Scanner.nextInt(Scanner.java:2076)
at TestScanner.main(TestScanner.java:9)
```

  - The source code can be compiled without any error
  - The error occurs when the program is running (**run-time error**)
    - In Java, it is called **exceptions**
    - There are many exceptions in Java (will get in more detail later)
- Check `ex4.java` for some examples about the `Scanner` class

Control Statements

- We already discussed some Java statements:
  - Declaration Statements:

```
int x;
double y;
String s;
```

  - Assignment Statements:

```
x = 5;
y = y * 0.2;
s = "Hello";
```

  - Method Calls:

```
System.out.println("Hello");
System.out.println(x);
```

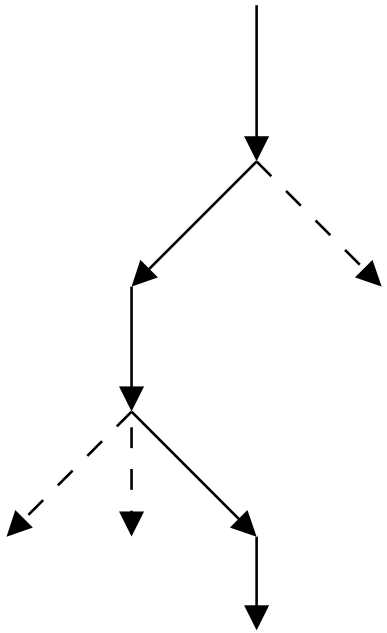
- Another important type of statements in programming is the **control statements**

Control Statements

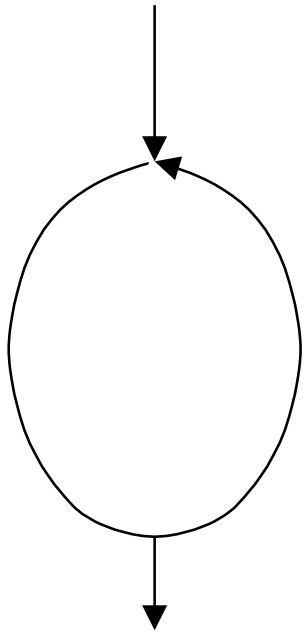
- A **control statement** allows two important type of execution
  - **Conditional execution**:
    - A statement or statements may or may not execute
  - **Iterative execution**:
    - A statement or statements may be executed zero or more times

Linear Execution

Conditional Execution



Iterative Execution



Boolean Expressions

- The key to many control statements in Java are **boolean expressions**
  - A **boolean expression** is an expression that is evaluated to either `true` or `false`
    - In Java, `true` and `false` are predefine literals
  - To create a boolean expression, we use one or more **relational operators** and **logical operators**

Relational Operators

- A **relational operator** is used to compare (relate) two primitive values
- Commonly used relational operators:
  - < (less than)
  - > (greater than)
  - == (equal)
  - != (not equal)
  - <= (less than or equal)
  - >= (greater than or equal)
- Examples:
  - `5 < 9` is evaluated to `true` since 5 is less than 9
  - `12 != 12` is evaluated to `false` since 12 is equal to 12

Logical Operators

- Some boolean expression are too complicate and required more than one relational operators
  - Example: To check whether `x` is greater than or equal to 0 and less than 12
    - `0 <= x < 12` is **NOT** a Java boolean expression
    - We need two boolean expressions, `0 <= x` and `x < 12`
    - A **logical operator** is needed to combine them into a single boolean expression
  - Frequently used logical operators are:
    - `!` (not),
    - `&&` (and), and
    - `||` (or)
  - Evaluations:

p	q	!p	p && q	p || q
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

  - From previous example, we need `0 <= x && x < 12`

Examples

- Variables are declared and initialized as follows:

```
int i = 10, j = 15, k = 20;
double x = 10.0, y = 3.333333, z = 100;
```

`true` or `false`?

- `(i / 3) == y)`
- `(x / 3) == y)`
- `(x / 3) > y`
- `x == i`
- `!(x != i)`
- `i < j || j < k && x <= y`
- `x <= y && y <= 12`
- `x <= y && y >= x`

- Precedence and Associativity
  - `!` has the highest precedence and right-to-left associativity
  - `&&` has higher precedence than `||` and both are left-to-right associativity

The `if` Statement

- The `if` statement is very intuitive:

```
if(boolean_expression)
    <true options>;
else
    <false options>;
```

- Each `<true options>` and `<false options>` can be any Java statement including a **block**
  - Java blocks are delimited by { and } and can contain any number of statements
- The `else` together with `<false options>` is optional
- Parentheses around the above `boolean_expression` are required
- Example:

```
if(x % 2 == 0)
    System.out.println(x + " is an even number.");
else
{
    System.out.println(x + " is odd, turning it to an even number.");
    x++;
}
```

Nested `if` Statements

- Since `<true options>` and `<false options>` can be any Java statement, they can be `if` statements
- This allows us to create **nested `if` statements**

```
if(...)
{
    :
    if(...)
    {
        :
    }
    else
    {
        :
    }
    :
}
```

where `...` are boolean expressions and `:` are Java statement(s)

Nested if Statements

- We can nest on <true options>, on <false options>, or on both
- Enable use to test multiple conditions and to have a different result for each possibility
- Example: Check whether 0 <= x < 10. If not, show that it is out-of-range. However, if it is in range, also check whether it is an even or odd number.

```
if(x >= 0 && x < 10)
{
    System.out.println("x is good");

    if(x % 2 == 0)
        System.out.println("x is even");
    else
        System.out.println("x is odd");
}
else
{
    System.out.println("x is out-of-range");
}
```

Dangling else

- Consider the following code

```
if(condition1)
    if(condition2)
        statement1
else
    statement2
```

  Which `if` statement that the `else` is associated with?

- Rule: An `else` will always be associated with the **closest unassociated** `if`
- If a programmer does not understand the rule, it will cause a `logic error`
  - Difficult to find and correct
  - Can be compiled without problem but gives incorrect results

Logic Error

- A **Syntax Error** prevents a program from being compiled
  - Error message(s) together with locations (line numbers) will give you some hints
- A **Run-time Error** stops the program as soon as an error occurs
  - Type of exception together with locations (line numbers) will give you some hints
- A **Logic Error** does not cause anything. Your program simply produces incorrect result(s)
- **When in doubt, always use** { and }

```
if(condition1)
{
    if(condition2)
        statement1
}
else
    statement2
```

The else-if Construct

- Multiple conditions with a different result for each possibility

```
if(condition1)
{
    :
}
else if(condition2)
{
    :
}
else if(condition3)
{
    :
}
else
{
    :
}
```

The `while` Statement

- The `while` loop is also intuitive:

```
while(boolean_expression)
    <loop body>;
```

- The `<loop body>` can be any Java statement or block
- Logic of the `while` statement:
  1. Evaluate the `boolean_expression`
  2. If the result is
     - `true`, execute `<loop body>` and go to step 3
     - `false`, skip to the next statement after loop
  3. Repeat (go back to step 1)
- A `while` loop is called an **entry loop** because a condition must be met to get in to the loop body

Practice

- Let's write a simple program that uses `if` and `while`
- **Specification**: This program must be able to calculate an average of a set of scores
- **Discuss**: questions and/or issues
  - How many scores are there?
    - Do we know this in advance?
    - What if we do not know in advance?
  - Do we have a valid range of a score?
    - Between 0 to 100?
    - Something else?
  - A score is an integer or a real number?
    - This will effect the type of variable(s)
  - Any special cases that we need to consider?
    - What if the number of scores is 0?
    - What to do if a score is out-of-range?

Practice (Template)

- In Java, the name of the class must be the same as `.java` file
- Suppose we want to call this program `AverageCalculator`
- Template for the file `AverageCalculator.java`

```
public class AverageCalculator
{
    public static void main(String[] args)
    {

    }
}
```

- Check `ex5a.java` and `ex5b.java`

Common Mistake

- Consider the following code:

```
while(i < 10);
{
    statement1;
    statement2;
    i++;
}
```

- See something strange?
  - The semicolon right after the boolean expression
- The above code behaves exactly the same as the following code:

```
while(i < 10)
{
}

statement1;
statement2;
i++;
```

The `for` Statement

- The `for` loop is more complicate
- Example:

```
for(int i = 0; i < max; i++)
{
    // will iterates max times
}
```

- Syntax of a `for` loop

```
for(initial_expression; loop_condition; loop_expression)
{
    <loop body>;
}
```

The `for` Statement

- `initial_expression`
  - Can be any Java statement expression
  - Will be execute only one time when the loop is first executed
- `loop_condition`
  - Must be a Java **boolean** expression
  - Evaluated **prior** to each execution of the `<loop body>`
    - If `true`, `<loop body>` is executed
    - If `false`, loop terminates
- `loop_expression`
  - Can be any Java statement expression
  - Evaluated **after** each execution of the `<loop body>`
- These expressions/condition make the `for` loop extremely flexible

The `for` Statement

- Let's write some programs:
  - For loop to sum the numbers from `n` to `m`
    - `n` must be less than or equal to `m`
    - Calculate `n + (n + 1) + ... + (m - 1) + m`
  - For loop to output powers of 2 from 0^2 to k^2 for some integer k
    - `k` must be greater than or equal to 0
- A `for` loop has an equivalent `while` loop

```
for(initial_expression; loop_condition; loop_expression)
{
    <loop body>
}
```

is equivalent to

```
initial_expression;
while(loop_condition)
{
    <loop body>
    loop_expression;
}
```

The `switch` Statement

- As we discussed, the `if` statement can be used in a multiple form
  - Nested in a `<true option>`
  - Nested in a `<false option>`
  - Nested in both
  - Each nested can also be nested, and so on
- Sometimes, choices are simple:
  - Integral range of values (e.g., 3, 4, 5, or 6)
  - Set of values (e.g., 4, 12, 2, 9, 3)
- It is easier and more efficient to use a `switch` statement instead of nested `if` statement
  - Be careful, you must use it correctly

The switch Statement

- Syntax of the switch statement

```
switch(int_expr)
{
    case constant_expr:
    :
    case constant_expr:
    :
    default:  // this is optional
    :
}
```

- int_expr is initially evaluated
- constant_expr are tested against the int_expr from top to bottom
  - First one to match determines where execution within the switch body begins
  - However, **execution will proceeds from there to the end of the block**

Example

- Consider the following code snippet:

```
int x = 2;

switch(x)
{
    case 1:
        System.out.println("One");
    case 2:
        System.out.println("Two");
    case 3:
        System.out.println("Three");
    default:
        System.out.println("Not One, Two, or Three");
}
```

The output is

```
Two
Three
Not One, Two, or Three
```

- What if we want to print just "Two"?

Example

- If we want the execution of the different cases to be exclusive of each other, we need to stop execution prior to the next **case**
  - Use the `break` statement

```
int x = 2;
switch(x) {
    case 1:
        System.out.println("One");
        break;
    case 2:
        System.out.println("Two");
        break;
    case 3:
        System.out.println("Three");
        break;
    default:
        System.out.println("Not One, Two, or Three");
        break;  // optional (last one anyway)
}
```

  The output will be just `"Two"`
- Check `ex6.java` and `ex6b.java` for the `switch` statement

Methods and Method Calls

- If programs are **short**, we can write them as one contiguous segment
  - The logic is probably simple
  - There are not too many variables
  - Not too likely to make a lot of errors
- For long programs
  - Logic is much more complex
  - A lot of variables, expressions, control and loop statements
  - Chances of having one or more **bugs** is higher
    - Find it is hard (the code is long)
    - Fix it is difficult (logic is complex)
  - Hard to break up into multiple segments so that multiple programmers can work together
  - If parts need to be modified or added, it is difficult with one large segment
  - If similar actions are taken in various parts of the program, it is inefficient to code them all separately
- Most of these problems can be solved by breaking our program into smaller segments

Methods

- **Method** (or function, or procedure, or subprogram):
  - A segment of code that is **logically separate** from the rest of the program
  - When **invoked** (i.e. **called**), control jumps from the `main()` (for now) to the method and it executes
    - usually with **parameters** (**arguments**)
  - When it is finished, control reverts to the next statement after the method call
- Rough example (in the same file and class):

```
public class Example
{
    public static void main(String[] args) {
        statement1;
        myMethod(5);  // call myMethod()
        statement2;   // back to here when myMethod() is finished
    }

    public static void myMethod(int x) {
        <myMethod's body>
    }
}
```

Methods

- Methods provide us with **functional** (or procedural) **abstraction**
- We do not need to know all of the implementation details of the methods in order to use them
  - We only need to know:
    - What **arguments** (parameters) we must provides
    - What the **effect of the method** is (i.e. what does it do?)
  - Implementation of a methods can be done in multiple way
    - For example, the predefine method `sort(Object[] a)`
    - There are multiple way to sort
  - This allows programmers to easily use methods that they did not implement
- Example: `System.out.println("I am Groot!!!");`
  - We do not need to know how produce a sequence of characters on the console
  - We need to supply a `String` to the method `println()`
  - The effect is the given `String` will be printed on the console screen

Primary Uses of Methods

- Act as a **function**, returning a result to the calling code
  - These methods are declared with **return types**, and are called within an assignment or expression
  - Examples:

```
x = inScan.nextDouble();
y = Math.sqrt(x) / 2;
```

- Act as a **subroutine** or **procedure**
  - Executing code but not explicitly returning a result
  - Declared to be `void` (return type)
  - Called as a separate stand-alone statement

```
System.out.println("I am Groot...");
```

Predefine Methods

- There are many predefine methods (check online API)
- Often called in the following way

```
ClassName.methodName(parameter_list)
```

  - `ClassName` is the class in which the method is define
  - `methodName` is the name of the method
  - `parameter_list` is a list of **zero or more** variables, literals, or expressions that are passed to the method
  - Example:

```
y = Math.sqrt(x);
```

  - These are called **static** or **class** methods
    - They are associated with a class, not with an object

Predefine Methods

- Some are called in the following way:

```
ClassName.objectName.methodName(parameter_list)
```

  - `objectName` is the name of a static predefined **object** that contains the method
  - Example:

```
System.out.println("I am Groot...");
```

    - `System` is a predefined class
    - `out` is a predefine object (PrintStream) within `System`
    - `println` is a method within PrintStream
  - These are **instance** methods (associated with an object)
    - Will discuss about these type of methods later
  - For now, let's focus on **static** methods

Static Methods

- What if we need to use a method that is not predefine?
- Just have to write it ourselves
- Syntax (subroutine/procedure):

```
public static void methodName(parameter_list)
{
    // method body
}
```

- Syntax (function):

```
public static retVal methodName(parameter_list)
{
    // method body
}
```

  - `retVal` can be a valid Java type (e.g., `int`, `float`, `String`)
  - When a method is not void, there **must** be a `return` statement

Simple Example

- A simple example

```
public class GrootMethod
{
    public static void sayGroot()
    {
        System.out.println("I am Groot...");
    }

    public static void main(String[] args)
    {
        sayGroot();

        for(int i = 0; i < 10; i++)
            sayGroot();
    }
}
```

- In Java, a static method can be located before or after the `main()` method
- Note that we call the method `sayGroot` without using the class name `GrootMethod`

Parameters

- What about the `parameter_list`?
  - It is a way to **pass value(s) into our methods**
  - Enables methods to process different information
    - More useful and flexible
  - The syntax of `parameter_list` is as follows:
    - `type identifier` pair(s) separated by commas

```
public static retType oneParam(type1 id1) {...}
public static retType twoParams(type1 id1, type2 id2) {...}
```

    - Called *formal parameters*, or **parameters**
  - In the **method call**:
    - List of variables, expressions, or literals that match one-by-one with the parameters in the method's definition
    - Both types and number of parameters must match
    - Called *actual parameters*, or **arguments**

Parameters

- Example

```
public class AddSome
{
    public static void add(int a, int b)
    {
        int result = a + b;
        return result;
    }

    public static void main(String[] args)
    {
        int x = 5;
        int y = add(x, 12);
        System.out.println(y);   // print 17
    }
}
```

- Again, if a method is called in the same class in which it was defined, no need to use the class name during call

Parameter Passing

- Parameters in Java are passed **by value**
  - The parameters is a **copy** of the evaluation of the argument

```
1  public class AddSome {
2      public static void add(int a, int b) {
3          int result = a + b;
4          return result;
5      }
6
7      public static void main(String[] args) {
8          int x = 5;
9          int y = add(x, 12);
10         System.out.println(y);   // print 17
11     }
12 }
```

1. Line 9: `x` is evaluated to 5
2. Line 2: `a` is initialized to 5 and `b` is initialized to 12
3. Line 3: `result` is set to `a + b` which is 17
4. Line 4: Return 17 back to the caller method
5. Line 9: `add(x, 12)` is evaluated to 17 (returned by the method)

Parameter Passing

- Parameters in Java are passed **by value**
  - Any changes to the parameter do not effect the argument

```
public class ByValue
{
    public static void changer(int a)
    {
        a = 100;
    }

    public static void main(String[] args)
    {
        int a = 1;
        changer(a);
        System.out.println(a);   // print 1
    }
}
```

Effect of Value Parameters

- Arguments passed into a method cannot be changed
- Pros:
  - Prevents accidental side-effects from methods
- Cons:
  - What if we want the arguments to be changed?
    - I want `swap(x,y)` to actually swap value in `x` and `y`

```
public static void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

    - Perform the following has no effect:

```
int x = 5, y = 12;
swap(x, y);
```

    - We can get around when we learn about object-oriented programming

Local Variable and Scope

- Variables declared within a method are **local to that method**
  - We usually called **method variables**
- They exist only within the context of the method
- Parameters are also local variables which are initialized during method call
- The scope of these variables is point in the method that they are declared up to the end of the method

Local Variables and Scope

- Example

```
1  public class Local
2  {
3      public static int aMethod(int a)
4      {
5          // x and args are not available in this method
6          double result;
7          int b = 2;
8          return Math.sqrt(result) / b;
9      }
10
11     public static void main(String[] args)
12     {
13         // a and b are not available in this method
14         double x = 12.345;
15         double result = aMethod(x) + 12;
16         System.out.println(result);
17     }
18 }
```

- Note: There are two different `result` variables

Local Variable and Block

- Java variables can also be declared within blocks inside of methods

```
1  public static int doSomething(int a, int b)
2  {
3      int x = 5;
4
5      if(a < b)
6      {
7          int x = 12, y = 9;
8          System.out.println(x);   // print 12
9      }
10
11     System.out.println(x); // print 5;
12     System.out.println(y); // Compilation error
13 }
```

- The scope is the point of the declaration until the end of the block
  - The scope of x at line 7 is from line 7 (point of declaration) to line 9 (}).

Local Variables and Scopes

- Again, local variables cannot be shared across methods
  - A local variable declared in one method cannot be accessed in a different method
    - If they have the same name, they are two different variables
  - We can still get data from one method to another
    - How?
  - To share variables across methods, we need to use object-oriented programming
- See `ex7.java`