Lecture 02: Introduction to Java

This chapter contains:

- Overview of Java
- Compilation Process
- Anatomy of a Java Program
- Outputting Information on Screen
- Primitive Data Types & Variables
- Operations
- Expressions
- Object-Oriented Programming Conventions
- Getting Input From Input Dialogs
- Getting Input From the Console
- The String Class
- The Character Class
- Formatted Output
- Keeping Track of Date and Time
- Java Keywords
- Commonly Used Java Packages
- Summary and Concluding Remarks
- Review Questions
- Recommended Readings

2.1 Overview of Java

Java is a purely object-oriented programming language (OOPL), originally developed by Sun Microsystems, and currently marketed by Oracle (after acquiring Sun Microsystems). The language, though relatively new, has been through several iterations of refinement. The currently available version is Java 8 Standard Edition (SE). Oracle markets various software packages including the Java language. The ones of immediate interest are:

- Java Development Kit 8u25
- Java SE 8u25
- Java SE Run-time Environment 8u25
- Java Development Kit 8.0 with NetBeans 8.0.2

In addition, Oracle markets several other Java-based products. Each product line is marketed with a comprehensive set of documentation, which is accessible via the Oracle website. There are several third-party Java development kits that are available. One such product is Eclipse. You can learn more about this product from the related software documentation.

Java is a unique OOPL several respects. Two unique features about the language are worth noting here:

- Java is the first programming language to offer platform independence. This is achieved through its *Java Virtual Machine* (JVM). This JVM allows the programmer to write Java code that can be executed on multiple operating systems (OS) platforms without and modifications.
- Java is comprehensive in its scope. It has features that are applicable in traditional programming, user interface design, Web development, and network programming.

There are some additional features of Java that are worth mentioning:

- Java is relatively simple, compared to C++, the language on which it was modeled. This is particularly noticeable in areas such as inheritance (the use of interfaces provides a more graceful treatment of multiple inheritance than in C++); elimination of pointers; treatment of arrays; etc.
- Java is ideal for distributed processing such as in a network environment. It is also widely used as an internet programming language.
- Java is interpretive. As you type in your program, you will know almost immediately (and well ahead of compilation) whether you have conformed to the required syntax.
- Java is a reliable language that is widely used in industry. It can be used to construct very secure systems.
- Due to its platform independence, Java is a very portable language. It can be used to build software that will run on completely different systems. This brings flexibility to the software engineering arena.
- Java is multithreaded. This means that the language can be used to develop applications that contain several simultaneous tasks. This feature significantly adds to Java's suitability for distributed processing, network applications, and graphical user interfaces (GUI).

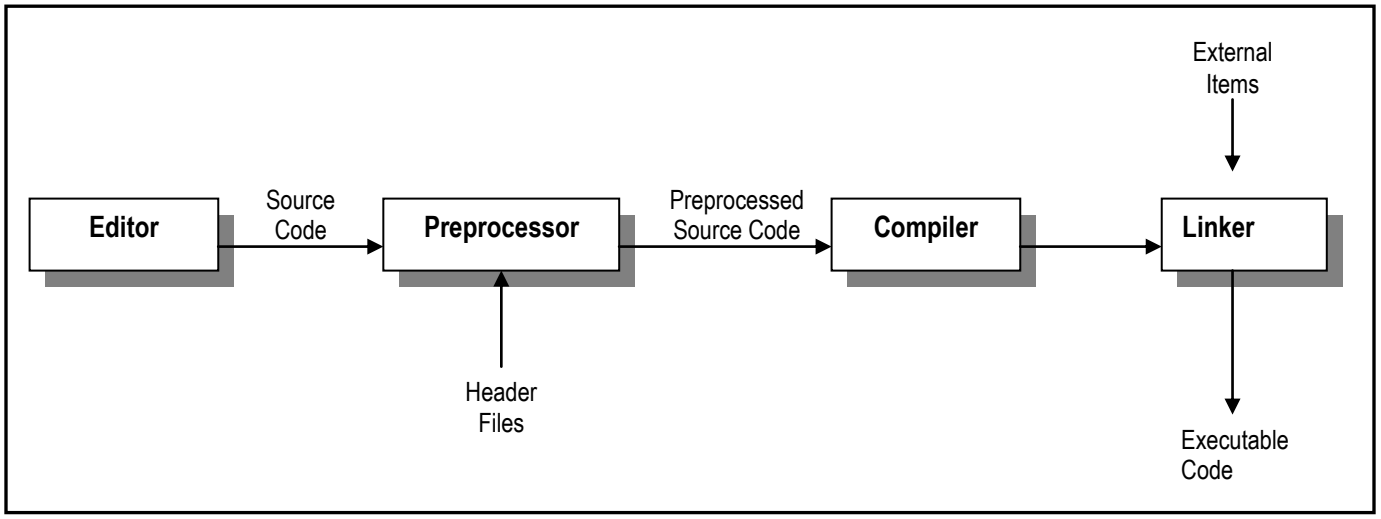
2.2 Compilation Process

Before we proceed further, it is imperative that you have an appreciation of what happens when you run a computer program: As mentioned above, Java has its roots from the C++ programming language. For C++ and other similar languages, a typical program passes through a number of important stages before it is executed by the computer. These steps are often referred to as the *compilation process* and include:

1. Source code entry via the *Language Editor*
2. Preparation for compilation via the *Preprocessor*
3. Compilation via the *Compiler* or *Interpreter*
4. Linkage
5. Execution

Figure 2.1 illustrates the interrelated components of a programming language environment and the various processes that the program passes through. Since this is a very important process, we will take some time to look at each major component in the figure.

Figure 2.1: The Compilation Process



2.2.1 Language Editor

The language editor (or simply the editor) is a program that allows the user (programmer) to key in his/her program (source code). The editor may be a traditional line editor, or a graphical editor (for this course, you will most likely be using a graphical editor); this affects to a large extent, your programming environment. Typically, it provides facilities for the following:

- Entering and editing the program;
- Loading a program (from disk) into memory;
- Compiling the program;
- Debugging the program;
- Running the program.

2.2.2 Preprocessor

The preprocessor is a program that removes all comments from the source code and modifies it according to directives supplied to the program.

2.2.3 Compiler or Interpreter

The compiler is a program that accepts as input, the preprocessed source code, analyzes it for syntax errors, and produces one of two possible outputs:

- If syntax error(s) is/are found, an error listing is provided.
- If the program is free of syntax errors, it is converted to object code (assembler language code or machine code).

An interpreter is similar to a compiler; however, there are three subtle differences:

- The interpreter works on a command-by-command basis. It stops at the first syntax error, and refuses to continue until the user either instructs it to ignore that particular command, or makes a correction to the command. The compiler, on the other hand, looks at the whole program and attempts to list all errors encountered.
- The interpreter has an interactive orientation, while the compiler has a batch orientation
- The interpreter is more responsive, but less efficient than the compiler. This is due to the fact that the compiler produces an object code, which is stored by the system for subsequent usage. The interpreter converts source to object each time the program is run. Modern interpretive languages are sometimes designed to circumvent this disadvantage by providing a compile option, which saves a persistent copy of the object code as a traditional compiler does.

Note:

1. If the preprocessed code is converted to assembler code, an assembler then converts it to machine code.
2. Machine code varies from one (brand of) machine to the other. Each machine (brand) has an assembler. Assembler language programming is particularly useful in system programming and writing communication protocols. An assembler language is an example of a low level language.

2.2.4 Linker

A linker (linkage editor) is a program that combines all object code of a program with other necessary external items to form an executable program. This is done in a manner that is transparent to the user.

2.2.5 Programming Environment

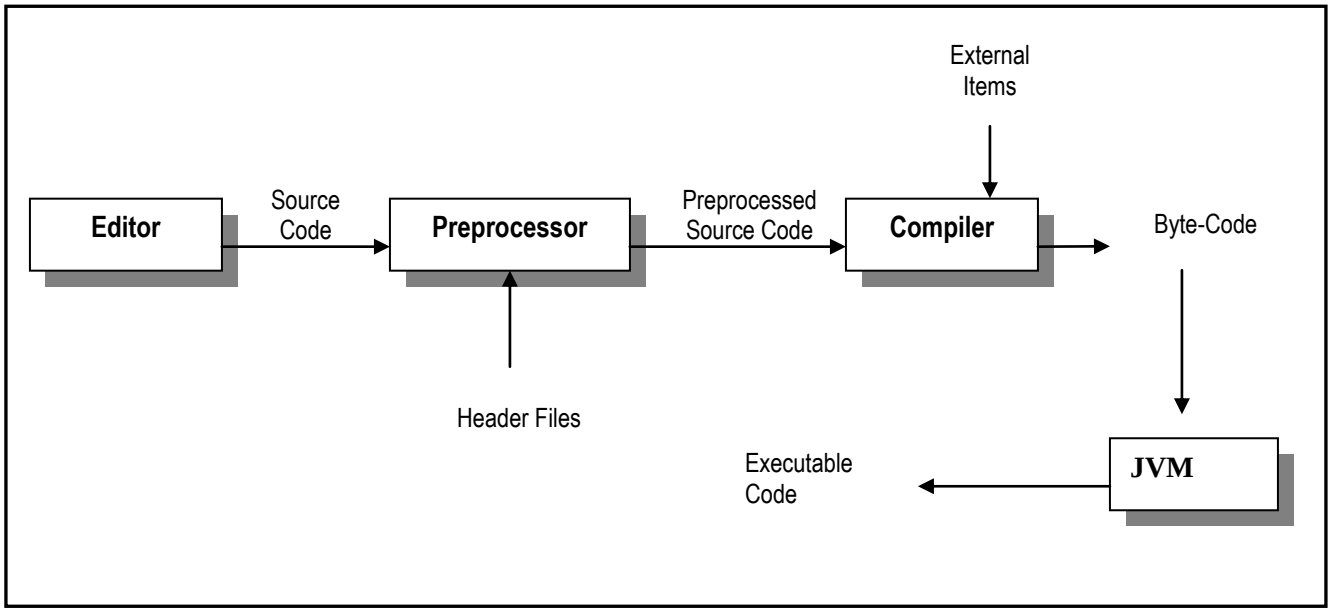
The typical programming environment is composed of the components shown in figure 2.1: the editor, preprocessor, compiler, linker, and a library of enhancement facilities that the programmer may find useful.

When you install a programming language, all these items are automatically included, and are loosely referred to as a compiler or a development kit. One obvious example is the Java Development Kit (JDK).

2.2.6 The Java Programming Environment

The Java programming environment is slightly different from the typical programming environment. This is intentionally so by design: In order to achieve the objective of platform independence, Sun Microsystems (brilliantly) introduced and intermediate level between source code and object code. This intermediate level is what is referred to as the Java Virtual Machine (JVM), or *Byte-code*. The JVM is the critical platform independent component that converts byte-code into native machine code. Figure 2.2 illustrates.

Figure2.2: Java Compilation Process



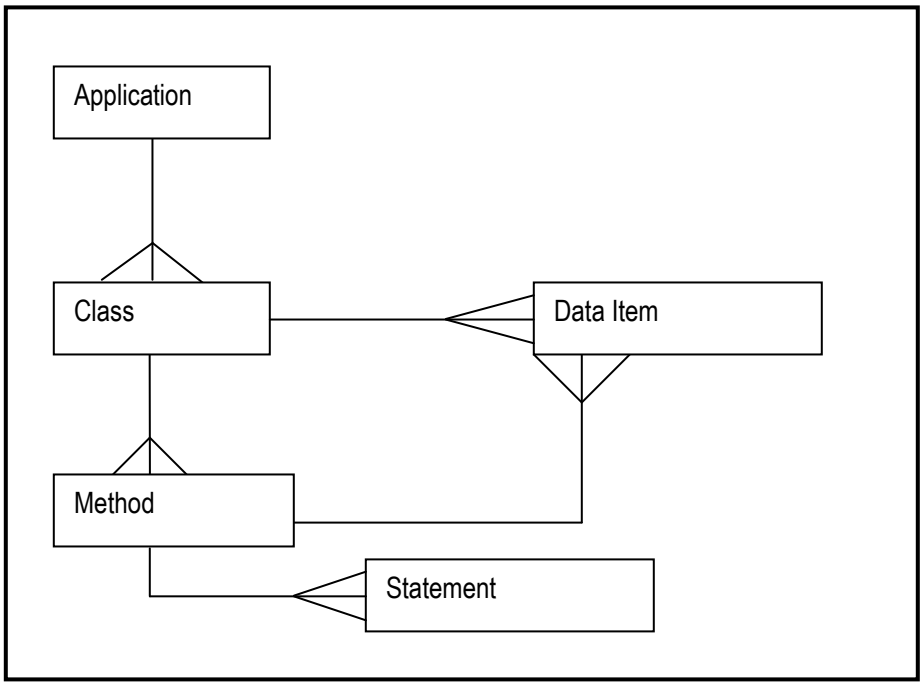
2.3 Anatomy of a Java Program

Very soon, you will be proudly writing Java programs. You must therefore know what a Java program looks like. Here is a summary:

- A Java program is essentially made up of one or more *classes*. The program is by definition, a class which may bring other classes into use. This is a loaded statement that will become clearer to you later in the course. For now, just consider a class to be a holding area for instructions.
- Classes are made up of one or more *methods* and *data items.*
- Methods are made up of instructions (in the form of *statements*) and possibly data items.
- Applications are made up of one or more classes; an application is implemented as a *package.*

Figure 2.3 provides a graphic illustration of the Java program anatomy. The multi-pronged lines (called "crow's feet") are used to convey the message that several referencing items (for instance classes) can comprise a referenced item (for instance application).

Figure 2.3: Java Program Anatomy



Note:

1. Each class is defined in a separate file with the name <ClassName>·Java. When the file is compiled, the object code is called <ClassName>· Class
2. The Java environment includes a number of built-in Classes that you can use. Writing Java applications is about learning to use these classes along with those that you create.
3. The angular brackets are used to indicate that you (the programmer) will supply the appropriate identifier. This convention will be consistently followed throughout the course, unless otherwise instructed.

2.3 Anatomy of a Java Program (continued)

Throughout this course, we will be using the Baccus-Naur Form (often called the BNF notation) for representing the syntax of Java programming language commands. The symbols used are depicted in figure 2.4.

Figure 2.4: BNF Notation Symbols

Symbol	Meaning
::=	"is defined as"
[. . .]	Denotes optional content (except when used for array subscripting)
<Element>	Denotes that the content is supplied by the programmer and/or is non-terminal
|	Indicates choice (either or)
{<Element>}	Denotes zero or more repetitions
<Element>*	Alternate notation to denote zero or more repetitions
<l>*<m><Element>	Denotes l to m repetitions of the specified element
[* <Element> *]	Alternate and recommended notation to denote zero or more repetitions for this course

Note: The construct {<Element>} is the original construct for repetition. However, C-based languages use the left curly brace ({) and right curly brace (}) as part of their syntax. To avoid confusion, it has been recommended that for these languages, the construct <l>*<m> <Element> or <Element>* be used. But that too is potentially confusing. Therefore, for this course, we will use the construct [* <Element> *] to denote zero or more repetitions.

In using this notation, there are two points of deviation that you need to bear in mind:

1. As mentioned in the figure, use of the originally symbols for repetition is problematic particularly for C-based languages (such as C, C++, Java, C#), due to the increased possibility of confusion for learners. To avoid this confusion, we will use the concoction [* …*] in this course to represent such repetitions.
2. There will also be exceptions with respect to the square braces and the angular braces; these will be pointed out at the appropriate time.

2.4 Outputting Information on Screen

There are several ways to output information to the screen in Java; however, in this course, we will examine two approaches: output via the console, and output via message dialogs.

2.4.1 Using the Console

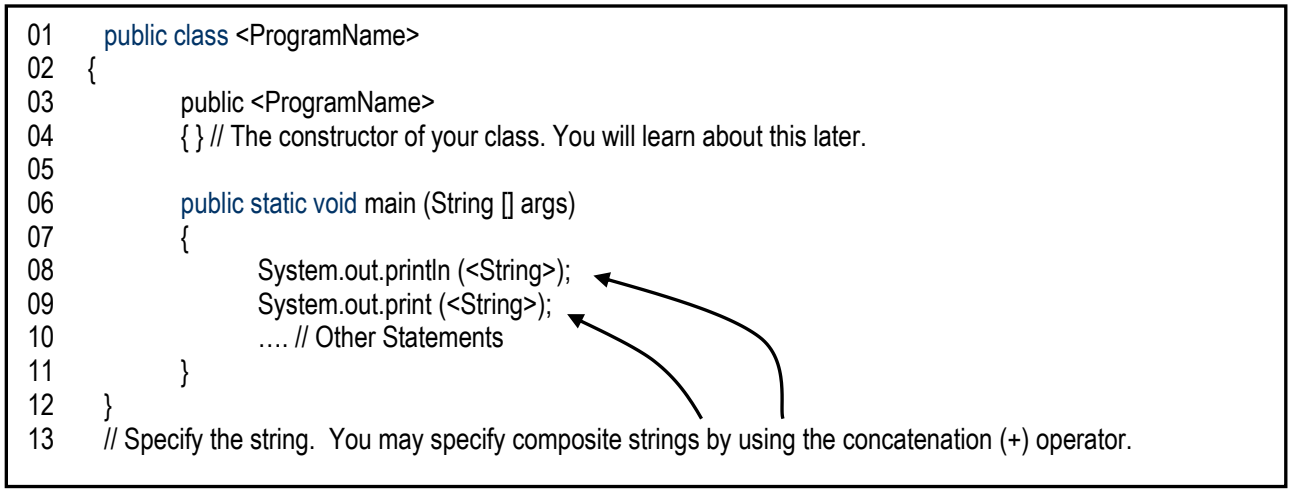
Figure 2.4 provides an illustration of a Java program to print a message on the screen. Let us spend a moment to explain this program structure:

- Line 1 is how you give your program a name. The keywords **public** and **class** are special reserve words in Java; you are not allowed to use them for unintended purposes. The keyword **public** means that the program (class) is going to be accessible to anyone intending to use the program; **class** defines your program as a class. You will learn more about these and other keywords later in the course. The line also requires you to specify a name for your program (class). The angular bracket simply means that instead of "ProgramName," you specify an appropriate name for your program.

2.4.1 Using the Console (continued)

- Line 2 contains the begin-block symbol and line 12 contains the corresponding end-block symbol for your class. All the code for your class must be inserted within the class-block.
- Lines 3 & 4 can be ignored for now. Each class has what is called a *constructor*. However, a main program does not require one. You will learn more about constructors later in the course.
- Line 6 is very important: Every main program must have a method called **main**. In defining **main**, you must use the keywords **public static void**. In fact, you must define main exactly as indicated. The reason for this will become clearer as you learn more about Java.
- Line 7 contains the begin-block symbol and line 11 contains the corresponding end-block symbol for the **main** method. All the code for your class must be inserted within the method-block.
- Lines 8 & 9 contain two method calls: **print** and **println** are two methods within the class **System.out** Each receives a string as argument, and displays this string to the console.

Figure 2.4: Illustrating a Basic Program Structure



Note:

1. The method **main** must be in your main program.
2. Java uses **System.out** to denote the screen and **System.in** to denote the keyboard. You will learn more about these later in the course.
3. Every class is defined in a separate file with the name of the class.
4. Every valid Java statement must end with a semicolon.

Figure 2.5 provides an example of a simple Java program that applies these principles. You will notice in this and almost every programming example, the appearance of what are called comments. Comments are clarifying statements that the programmer includes throughout his/her code. They are not considered by the compiler to be part of the code; in fact, they are completely ignored by the compiler. However, their inclusion is highly recommended to provide clarity and readability to the program. Comments are specified in one of two ways as shown below:

- // This is a comment, which allies to the exact line or section of a line where it appears.
- / * This is also a comment. Use this format where your comment may run for more than a single line. Everything within this comment block is ignored by the compiler. */

Figure 2.5: A Simple Java Program That Outputs to the Console

```
// ************************************************************************
// Program: MainProg
// Author: Elvis Foster
// Created on January 9, 2005, 6:34 PM
// ************************************************************************
package javaapplication1;

public class MainProg
{

   /** Constructor  */
   public MainProg()
   {  } // A constructor is normally not required for a main program.

   public static void main(String[] args)
   {
      System.out.println("Welcome to Java World");
      System.out.println("This is my main method. Every main program needs one");
      System.out.print("Every class is defined in a separate file. ");
      System.out.println("The name of the class file is <classname>.java");
   } // Ends the main method
} // Ends the class
```

2.4.2 Using Message Dialogs

There will be times when you want to display information in a much more attractive manner than **System.out.print()** is able you to do. Java provides you with several more sophisticated ways to do this, and as you get into Java graphics and Java user interfaces, you will learn these. However, there is another very simple, but more elegant way to display information via message dialogs. Figure 2.6 illustrates the basic syntax required and figure 2.7 provides an example:

Figure 2.6: Displaying Information via Message Dialogs

```
import javax·swing·JOptionPane; // JObtionPane is a Java class within a package called javax·swing
// ...

JOptionPane·showMessageDialog (<ParentComponent>,<Messages>,[<Title>, <icon>]);
```

A method of the JOptionPane class

Usually **null**

A String

A String

Icon selected from a set of predefined values.

Figure 2.7: A Simple Java Program That Outputs via a Message Dialog

```
// ************************************************************************
// Program: Third
// Author: Elvis Foster
// ************************************************************************

package javaapplication1;
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.

public class Third { // Beginning of the class

  public Third()
  { } // Null Constructor; we will discuss constructors later in the course
   public static void main(String[] args)
  {
   /* There are two types of message dialogs. One requires parent component (usually null), the message,
      the message box title, and the message box icon. The other requires the first two arguments only. */
   JOptionPane.showMessageDialog(null, "Welcome to Java World", "Enjoy Java", +
   JOptionPane.INFORMATION_MESSAGE);
   JOptionPane.showMessageDialog(null, "This is my third Java class. Please note the following:" +
   "\n" + "A Java program is made up of classes." + "\n" + " A class consists of one or more methods. " +
   "The class may also contain data items." + "\n" + " An application typically consists of several classes.
   " +  "\n" + "Applications are implemented as packages.");
  }
} // End of the class
```

Please observe:

- Within the Java class **javax.swing.JOptionPane**, there is a method called **showMessageDialog()** that requires the arguments shown. This method displays a dialog box with the specified message being displayed.
- In order to use this method, you import the **javax.swing.JOptionPane** class into your program via the **Import-Statement**. The statement is coded in your program by simply specifying the import keyword, followed by the name of the Java package or class being imported. You can also use the wildcard (*) in the statement to include all components in that particular package. For instance, the following statement is used to import all resources in the **java.util** package.

```
import java.util.*;
```

2.5 Primitive Data Types & Variables

So now you know how to write a very basic Java program that prints information on the screen. Nice! But you will need to learn a lot more than that. Next, you want to be able to define basic variables, read them into your program, manipulate them, and perhaps produce more interesting outputs. Yes? In order to do so, you need to learn how Java represents the primitive data types, and how to define variables on them.

2.5.1 Primitive Data Types

Every programming language has primitive data types — the basic types of data that are stored and upon which basic operations are carried out. The primitive data types of Java are similar to those of C++ (the language from which Java has a strong reference, and which was used to develop Java). The primitive data types of Java are:

byte	short	int	long	float	double
char	boolean	void	String	enum	

Figure 2.8 provides relevant information about each type. You do not have to memorize this information; however, you must bear it in mind as you write programs involving these data types.

Note:

1. **String** is implemented as a class, not a primitive data type (the Java convention is to begin class-names with an upper-case letter). However, it is listed here for two reasons:
   - In some languages, the string is implemented as a primitive type.
   - In all languages (including Java), strings are widely used as the means of getting basic data to and from the program. As you will soon see, early mastery of strings is essential to your programming journey with Java.
2. The **enum** type is a special type that also results in the implementation of a class. A discussion of this is deferred until chapter 4.

Figure 2.8: Primitive data Types

Type	Range	Comment
byte	-2^7 to $2^7 - 1$	Storage is 8-bit signed.
short	$-2^{15}$to $2^{15} - 1$	Short integer. Storage 16-bit signed.
int	$-2^{31}$to $2^{31} - 1$	Integer. Storage 32-bit signed.
long	$-2^{63}$to $2^{63} - 1$	Long integer. Storage 64-bit signed.
float	-3.4E38 to 3.4E38	Floating point number (6-7 significant digits of accuracy). Storage 32-bit; IEEE754 standard for floating point numbers.
double	-1.7E308 to 1.7E308	Double precision floating point number (14-15 significant digits of accuracy). IEEE754 standard for floating point numbers.
char	2^8 (i.e. 256)	Storage is 8-bit unsigned.
boolean	**true** or **false**	Boolean value.
void		Special keyword used to indicate that a method does not return a value.
String	Variable length	Implemented as a Java class, not a primitive type.
Note: Primitive type names are reserve words. So are **true** and **false**.		

2.5.2 Variables and Identifiers

An *identifier* is a name that is given to a Java program element (object). This object may be a class, an instance of a class, a method, a constant, a package, or a variable. Below are some basic rules for naming identifiers:

- The identifier must start with a letter, an underscore, or a dollar sign ($); it is highly recommended to start your identifiers with a letter. The Java convention is to begin identifiers (variable-names and method-names) with a lower-case letter. However, it improves the readability of your code if you slightly deviate from this and begin all identifier names with an upper-case letter. Identifiers for constants are usually stated in upper case.
- The identifier cannot be a Java reserve word.
- The identifier may be of any length.

You were introduced to the concept of variables in the previous lecture. From a programming perspective, a variable is a data item that belongs to a particular primitive data type. Its value may change at any time during the execution of the program. Variables are declared as identifiers. In fact, the terms *variable* and *identifier* are sometimes used interchangeably. The required syntax for declaring Java variables is as id shown in figure 2.9a:

Figure 2.9a: Syntax for Variable Declaration

VariableDeclaration ::=
[<Qualifier(s)>] <DataType> <VariableName> [= <Expression>] [* ,<VariableName> [= <Expression>] *];

Please observe:

1. Qualifiers are special reserve words that are used to influence how the variable is to be used. Figure 2.9 provides a list of commonly used qualifiers and their meaning. You may use more than one qualifier on a variable declaration.
2. The data type specified must be a valid primitive type, or an advanced (programmer-defined) type. For now, we will concentrate on primitive types.
3. Each variable-name specified is by definition, an identifier, and must therefore follow the rules for identifiers.
4. An expression is as described in the previous lecture – a phrase that evaluates to a specific value. It can be a variable, a literal, or a combination of variables and operators. We will revisit expressions later in this lecture.
5. The act of specifying an expression with variable declaration is called *variable initialization*.
6. A constant is defined by using the **final** qualifier, and initializing the identifier.

Figure 2.9b: Commonly Used Java Qualifiers

Qualifier	Explanation
public	The item is accessible from anywhere in the program for instance, through instances of the class.
private	Only methods of the class can access the item.
protected	This item will be protected within the class hierarchy. It will be further elaborated in lecture 6.
static	This resource (data item or method) can be used directly without an instance of the parent class being created. In such case the class-name often takes the place of the instance name, when reference is made to that resource.
final	The item is a constant. It value will not change for the duration of the program.
abstract	This will be discussed later in the course.
The only four qualifiers you need to concern yourself with at this point are **public**, **private**, **final** and **static**.	

2.5.3 Variable Scope and Initialization

All variables declared in the class, ahead of the methods are referred to as *global* variables. They are accessible to all methods within the class. As you will see later, you can actually make a global variable accessible from outside of the class by using the **public** modifier. You will learn more about modifiers in subsequent lectures.

Variables that are declared within a method, or within a specific block of a method, are sometimes referred to as *automatic* variables. They are known only within the program block that they have been defined. It is very important that you understand this concept, and it will be reinforced several times throughout the course.

At the point where a variable is declared, you have the option of initializing it to some value. If you do not initialize the variable at declaration, it will be automatically initialized to a default value (**null** for strings and characters, or zero for numeric variables).

Example 1: Figure 2.10 illustrates variable declaration, initialization and other related issues.

Figure 2.10: Illustrating Primitive Types, Variables & Constants

```
// ************************************************************************
// Program: Demo
// Author: Elvis Foster
// Created on June, 2005
// ************************************************************************
package javaapplication1;

public class Demo
{ // Beginning of Demo Class
   // These data items (variables) are known to all methods of the class; they are global
   String stringA, stringB;
   int dateOfBirth, studNumber = 2004001; // studNumber has been initialized
   final String INSTITUTION = "Keene State College"; // This is a constant; the convention is to use upper-case on constants

  public Demo()  { } // Null Constructor

   public static void main(String[] args)
  { // Beginning of main
   // Any variable that you declare within this block is known only to the main method. They are
   // not accessible to any other method.
   // ...
         {
          // Any variable declared within this block are known only to the block.
         }
  } // End of main

  public void AnotherMethod()
  { // Beginning of AnotherMethod
    // Any variable declared within this method is known only to this method.
  } // End of AnotherMethod

} // End of Demo Class
```

2.6 Operators

There are two categories of operators in Java (and in most programming languages) — arithmetic operators and relational (Boolean) operators. We will discuss arithmetic operators here, introduce Boolean operators, but leave a full discussion of them for the next chapter.

2.6.1 Arithmetic Operators

Arithmetic operators are used in arithmetic expressions. Figure 2.11 lists the commonly used arithmetic operators, including the so-called *shortcut operators*.

Figure 2.11: Commonly Used Arithmetic Operators

Regular Operators	
Operator	**Meaning**
+	Addition or concatenation
-	Subtraction or negation
*	Multiplication
/	Division
%	Modulus (remainder of integer division)
++	Increment by 1 (may be prefix or postfix as illustrated below) counter++ // increment after use ++counter // increment before use
--	Decrement by 1 (may be prefix or postfix)
=	Assignment. Thus counter = counter+1 is equivalent to ++counter od counter++
Shortcut Operators	
Operator	**Meaning**
+=	Addition and assignment as illustrated below: x += y is equivalent to x = x + y
-=	Subtraction and assignment as illustrated below: x -= y is equivalent to x = x – y
*=	Multiplication and assignment as illustrated below: x *= y is equivalent to x = x * y
/=	Division and assignment as illustrated below: x /= y is equivalent to x = x / y
%=	Modulus and assignment as illustrated below: x %= y is equivalent to x = x % y
Generically, if x and y are variable, then x <RegularOperator>= y is equivalent to x = x RegularOperator y	

2.6.2 Boolean Operators

Boolean operators are used in relational (Boolean) expressions. Figure 2.12 lists the commonly used relational operators. These Boolean operators are used in Boolean expressions, which will be further discussed in the next lecture.

Figure 2.12: Commonly Used Boolean Operators

Operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal to
||	OR
&&	AND
!	NOT
&	Unconditional (bitwise) AND
|	Unconditional (bitwise) OR
?:	Conditional operator (this is a ternary operator, requiring three operands)

2.7 Expressions and the Assignment Statement

In basic algebra, you use variables, literals and operators to form mathematical expressions which can be evaluated and/or put to use in other expressions and ultimately in equations. A similar situation occurs in computer programming: we use variables, literals, and operators to form expressions which are then put to use in other expressions or assignment statements. As you will soon see, in many programming languages (Java included), the assignment statement is also an expression.

There are three types of expressions that you must be familiar with: arithmetic expressions, assignment statements, and Boolean expressions. We will discuss the first two here, and defer Boolean expressions for the next lecture.

2.7.1 Arithmetic Expressions

The syntax for an arithmetic expression in Java is shown in figure 2.13. As you study the figure, please note the following:

1. From this definition, observe that an expression may have several different formats. As an exercise you should try to identify them.
2. A rule often used to shorten code is to specify an assignment as part of a larger expression.
3. Parentheses when used, take higher precedence than any other operator.
4. Make sure that the operands used for binary operations belong to the same data type.

Example 2: The lower part of figure 2.13 provides some examples of valid and invalid expressions.

Figure 2.13: Arithmetic Expression in Java

ArithExpression ::=
<Literal> | <Variable> | <ShortcutExpression> |
<IncDecOpr> <Variable> |
<Variable> <IncDecOpr> |
[<ArithExpression> <ArithOperator> <ArithExpression>]

ShortcutExpression ::= <Variable> <Operator> = <Expression>

ArithOperator ::= + | - | * | / | % | = **IncDecOper** ::= ++ | --

Clarification on Shortcut Expression:

<Variable> = <Variable> <Operator> <Expression> // may be shortened to
<Variable> <Operator> = <Expression>

// Examples
x = x + y; /* is equivalent to */ x += y;
x = x – y; /* is equivalent to */ x -= y;
x = x * y; /* is equivalent to */ x *= y;
x = x % y; /* is equivalent to */ x %= y;
x = x / y; /* is equivalent to */ x /= y;

// Assume the following declarations: int x, y, z; String stringA, stringB;	
x = x * 9 * y;	Valid. x takes on the value 9xy
x = y % x;	Valid
x = y – 8 * x * x;	Valid
z = (9 * y – x) * 17;	Valid
stringA = stringB * x;	Invalid. Cannot mix operands of different data types
stringA = stringB + “the Great Pretender”;	Valid. The + acts as the concatenation operator
x - stringB	Invalid. Cannot mix operands of different data types
x * y - z	Valid expression, but is not a statement.

2.7.2 Assignment Statement

From the above definition and examples, it should be clear to you that an assignment statement is simply a special expression that involves the assignment operator (and terminates with a semicolon). Typically, your program will contain several assignment statements. In fact, whenever you wish to assign value to a variable, you do so by specifying an assignment statement. Figure 2.14 expresses this in BNF notation.

Example 3: Examine the Java code in figure 2.15 and figure out what it is doing. If you are struggling to explain the code, then go over the previous sections. If you got it at the first attempt, then congratulations!

Figure 2.14: Assignment Statement

```
AssignmentStatement ::=
<Variable> = <Expression>; |
<IncDecOpr> <Variable>; |
<Variable> <IncDecOpr>;
```

Figure 2.15: Illustrating Variable Declarations, Expressions, Assignments, and Output Display

Examples: Try reading this Java code and explain what it is doing:

```
float x, y; x = y = 0.00;
final int xCOEFF = 6;
final int xxCOEFF = 4;
final int CONST = 10;

final String LOUSYDAY = "This is a cranky day and it is snowing.";
final String LOVELYDAY = "This is a beautiful day; the sun is out and the temperature is about 70⁰F.";
final String PROGRAMHEADING = "The Ranting of Bruce Jones.";
String dayCommentry1, dayCommentry2, mathCommentry;

// . . .

z = xxCOEFF * x * x + xCOEFF * x + CONST; // y = 4x² + 6x + 10
dayCommentry1 = LOUSYDAY + " " + "I want to go home!";
dayCommentry2 = LOVELYDAY + " " + "A perfect day for some tennis!";
mathCommentry = "Here is the result of the function y = Ax² + Bx + C, when the x² coefficient is " + xxCOEFF +
  " and the x coefficient is " + xCOEFF + "and the constant is " + CONST + ": " + y;

 // . . .

 JOptionPane.showMessageDialog(null, dayCommentry1, PROGRAMHEADING, +
    JOptionPane.INFORMATION_MESSAGE);
JOptionPane.showMessageDialog(null, dayCommentry2, PROGRAMHEADING, +
    JOptionPane.INFORMATION_MESSAGE);
JOptionPane.showMessageDialog(null, mathCommentry, PROGRAMHEADING, +
    JOptionPane.INFORMATION_MESSAGE);

// . . .
```

2.7.3 Java Data Representation and Escape Characters

Java supports the Unicode coding system for data representation. This system employs a 16-bit code for each character. Unicode supports all the ASCII characters. Additionally, it is flexible and extensible enough to also support other special characters that are not easily represented in ASCII. Sun Microsystems's decision to support Unicode instead of just ASCII is also consistent with its decision and ambition to make Java a platform independent, ubiquitous programming language.

Java provides special codes for various escape characters that you will notice on your keyboard. Figure 2.16 provides a list of these special characters and their Java representation.

Figure 2.16: Java Escape Characters

Character	Java Representation
Backspace	\b
Tab	\t
Linefeed	\n
Return	\r
Form feed	\f
Backslash	\\
Single quote	\’
Double quote	\”

2.7.4 Data Conversion

Generally speaking, it is good programming habit to ensure that variables used in an expression belong to the same (or an agreeable) data type. In some instances, if they do not, you will get a program syntax error; in other instances, data conversion occurs. Let us briefly examine the latter case. Java facilitates data conversion in three possible ways:

Promotion Conversion: In these conversions, the narrower data type is converted to the wider data type (thus ensuring no data loss). To illustrate, if **payment** is a floating point variable, **hours** is an integer, and **paymentRate** is another floating point number, then the following statement will force a conversion of **hours** to a floating point number.

```
int hours; float payment, paymentRate;
payment = hours * paymentRate;
```

A similar conversion takes place when numbers are concatenated with strings: the numbers are converted to strings before the concatenation takes place. Please note that the conversion has no effect on the original definition of variables involved, only the implementation instances.

Assignment Conversion: An assignment conversion occurs when a value of one data type is assigned to a variable of another type. To illustrate, if **hours** is an integer variable, and **hoursWorked** is a floating point variable, consider the following statement:

```
float hoursWorked; int hours;
// …
hoursWorked = hours;
```

This statement will force a conversion of **hours** to a floating point number, and then assign its value to the variable **hoursWorked**. Like promotion conversions, assignment conversions have no effect on the original definition of variables involved, only the implementation instances.

2.7.4 Data Conversion (continued)

Casting: The third method of data conversion is referred to as *casting*. In this approach, you (the programmer) explicitly force a data conversion in order to satisfy the requirements of the situation at hand. A cast is a Java operator that is specified by a parenthesized data-type name, placed in front of the value to be converted.

Example 4: The above two illustrations could have been stated with explicit casting as follows:

```
int hours; float payment, paymentRate, hoursWorked, overtimeWorked;
// …
hours = (int) hoursWorked + (int) overtimeWorked; // Cast required
payment = (float) hours * paymentRate; // Cast not required
// …
hoursWorked = (float) hours - overtimeWorked ; // Cast not required but clarifying
```

As your knowledge of Java increases, you will discover many cases where casting becomes necessary, and not just optional. During this course, you will get a chance to go over some scenarios where casting is warranted. For now, you simply need to understand how it is done, and the examples provided will suffice.

2.7.5 Operator Precedence

As you write expressions, it will be imperative that you are aware of the precedence of the operators. Java implements the operator precedence rules as represented in figure 2.17. You will become more comfortable with this precedence table as you write more expressions in different programs.

2.17: Basic Java Operator Precedence Hierarchy

Priority	Operators	Clarification	Associativity
1	[ ]	Array index	Left to Right
	()	Parentheses or method call	
	.	Dot operator for member access	
2	++	Prefix or postfix increment	Right to left
	--	Prefix or postfix decrement	
	+ -	Unary plus or unary minus	
	~	Bitwise NOT	
	!	NOT	
	(type)	Type cast	
	new	Object instantiation	
3	* / %	Multiplication, division, and modulus	Left to right
4	+ -	Addition and subtraction	Left to right
	+	String concatenation	
5	<<	Signed bit shift left	Left to right
	>>	Signed bit shift right	
	>>>	Unsigned bit shift right — zero extension	
6	< <=	Less than; less than or equal to	Left to right
	> >=	Greater than; greater than or equal to	
	instanceof	Reference test — often used in instance casting	
7	==	Is equal to	Left to right
	!=	Is not equal to	
8	&	Bitwise AND	Left to right
9	^	Bitwise XOR	Left to right
10	|	Bitwise OR	Left to right
	|	Boolean OR	
11	&&	Boolean AND	Left to right
12	||	Boolean OR	Left to right
13	? :	Conditional operator which is ternary	Right to left
14	=	Assignment operator	Right to left
	*= /= += -= %= <<= >>= >>>= &= ^= |=	Shortcut operators	
Note: Highest priority is 1; lowest priority is 14.			

2.8 Object Oriented Programming Conventions

As you are aware, Java is a purely object-oriented programming language (OOPL). You are not going to get very far with it unless you get this concept clearly. Object-oriented programming (OOP) has certain basic conventions and principles that you must understand and master, if you are going to do well in the field. Below are some basic principles that you need to understand immediately. As we proceed through the course, we will revisit these and add more clarity, as well as introduce new ones.

Object Type & Object: An *object type* is a concept or thing about which data is stored. An object is an instance of an object type. For example, if **Student** is an object type, then we may observe that some student, **Bruce Jones**, as an instance of the object type **Student**. The simplest form of an object type is a primitive data type. The variables that you define on that data type are instances of that type.

Operation: An *operation* is a task that can be performed on an object. In an OOPL environment, operations are typically implemented as methods or functions (Java favors the term methods).

Example 5: We may define operations to be performed on a **Student** instance as follows:

> For the **Student** object type, valid operations may be **Add, Modify, Remove, Search, Display,** and **Print**.

Method: A *method* is a set of instructions for carrying out an operation. In some programming environments, methods are referred to as procedures and/or functions. Also, because a method merely implements an operation, the two terms are often used interchangeably. An older term which a method epitomizes is the *subroutine* (as discussed in lecture 1). However, Java favors the term method, so for the rest of this course, we'll stick with that.

Class & Encapsulation: In the simplest form, encapsulation is the act of hiding detail, of an object (type) until it is required. A *class* is the *encapsulation* of an object's structure with its operations. The object can be accessed only through its class. At the highest level, therefore, one is not focusing on one an object's (encapsulated) methods, but on some aspect of the object's structure, some of its operations, or both.

Classes & Methods versus Object Types & Operations: The convention in software engineering is to use terms such as classes & methods at the implementation level, and object types & operations at the design level. Since programming is predominantly an implementation issue (but you still have to design your programs), we will stick to the classes & methods.

Structure: The structure of a class refers to the data items and methods that have been defined within the class. The data items and methods are also referred to as the *properties* of the class.

Amalgamation: An object may be composed of other objects. In OOP, we refer to such an object as an *aggregate* or *composite* object. The act of incorporating other component objects into an object is called *aggregation* or *composition*. Since this is done through the object's class, the class is also called an aggregation (or composition) class. Throughout this course, we shall use the term amalgamation to mean an aggregation and/or composition.

2.8 Object Oriented Programming Conventions (continued)

Inheritance: An object inherits all the *properties* of its parent class**.** Additionally, a class may inherit from another class. The inheriting class is referred to as the *sub-class* and the inherited class is called the *super-class*.

Example 6: Here is an example that combines many of the concepts previously discussed:

Computer-Science-Student could be defined as a sub-class of the super-class **Student**; **Rectangle** could be defined as a sub-class of the super-class, **Polygon**.

- If **Bruce** is a **Student** object, **Bruce** inherits all the inheritable properties of **Studen**t.
- If **Karen is Computer-Science-Student** object, **Karen** inherits all the inheritable properties of **Computer-Science-Student** and **Studen**t.
- The **Student** object may be composed of a **StudentPersonal** object and a **StudentAcademic** object. If so, then **Student** is an amalgamation.

Polymorphism: An operation may be required to perform differently, depending on the context of its usage. This phenomenon is called polymorphism. In Java, polymorphism is implemented via *method overloading* and *method overriding*. You do not need to worry about this now; we will revisit it later.

The Dot Operator: In OOP, whenever we want to refer to a component of a package, or class, we use the dot operator (which is simply a period). You will see the dot operator used a lot throughout this course, and will no doubt use it in your own programs.

Example 7: Following is an illustration of how the dot operator is used:

Suppose that **Student** is a class with data items **name** and **dateOfBirth**. Suppose further that it also has methods **addMe() and printMe()**. If **anyStudent** is an instance of **Student**, then we can refer to its components as **anyStudent.name**, **anyStudent.dateOfBirth**, **anyStudent.addMe()** and **anyStudent.printMe()**.

The dot operator is also used to refer to Java components. For instance, **java.lang.Integer** refers to a class called **Integer** in the Java package called **java.lang**, while **javax.swing.JOptionPane** refers to a class called **JOptionPane** in the Java package called **javax.swing**.

Referencing Static Components: The concept of static resources (data items or methods) was mentioned earlier (section 2.4). The convention for referencing static resources is slightly different (due to the definition of a static resource): To reference a static resource, instead of specifying the instance name (there is no instance), you specify the class name where that static resource resides.

Example 8: Here is another illustration:

Suppose that **College** is a class with a static data item called **cName**. Then we would reference this data item as **College.cName** from outside of the class, and simple c**Name**, if we are referencing from within the class.

2.9 Getting Input From Input Dialogs

The simplest way to get input to your program is to use the **showInputDialog** method from the **Javax.swing.JOptionPane** class. It can be invoked in the following way:

Figure 2.18: Using the showInputDialog() Method

```
<StringVar> = JOptionPane.showInputDialog(<ParentComponent>, <PromptString>, <DialogTitle>, <Icon >);
// The ParentComponent is usually null. The Icon is usually JOptionPane.QESTION_MESSAGE
```

Please note: The input from a dialog box is always a string. To convert it to an integer, you can use the method **Integer.parseInt()** (i.e. method **parseInt()** from class **Integer** as follows:

```
<intVariable> = Integer.parseInt(<String>);
```

This is your first introduction to the **Integer** class! Actually, each primitive data type has a corresponding class. These classes are stored in the package **java.lang**. In each case, the class contains various methods, including those responsible for converting a string to other primitive types. Figure 2.19 provides the relevant information.

Example 9: Figure 2.20 illustrates a simple program to manipulate a parabola. The program prompts the user to specify the coefficients for x^2 and x, the x-value, as well as the constant. It then uses the formula $y = Ax^2 + Bx + C$ to determine the y-value for the input received, and reports the information to the user.

Figure 2.19: Corresponding Class for Each Primitive Type

Primitive Type	Class	Data Conversion Methods
byte	Byte	parseByte(String s) parseByte(String s, int radix)
boolean	Boolean	parseBoolean(String s)
char	Character	valueOf(char c)
double	Double	parseDouble(String s) shortValue() longValue() valueOf(double d) valueOf(String s)
float	Float	parseFloat(String s) shortValue() longValue() valueOf(float f) valueOf(String s)
int	Integer	parseInt(String s) parseInt(String s, int radix) shortValue() longValue() valueOf(String s) valueOf(int i)
long	Long	parseLong(String s) parseLong(String s, int radix) shortValue() longValue() valueOf(String s) valueOf(long l)
short	Short	parseShort(String s) parseShort(String s, int radix) shortValue() longValue() valueOf(String s) valueOf(short s)

Figure 2.20: Java Code for Parabola Manipulation

```
// ************************************************************************
/* Parabola.java                                                        */
/* Created on January 10, 2005, 12:30 PM                                */
/* Author Elvis Foster                                                  */
// ************************************************************************
package javaapplication2;
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.

public class Parabola
{
   public static void main(String[] args)
   { // Start main Method

         // Class Assignment #1: This program manipulates some strings
         int coeffA, coeffB, constC, varX;
         String inputA, inputB, inputC, inputX, outputString1, outputString2;
         float varY;
         final String HEADING = "Parabola of Bruce Jones";

         // Accept the names of the visitors:
         inputA = JOptionPane.showInputDialog(null, "Specify the Coefficient A: ", HEADING,  +
                  JOptionPane.QUESTION_MESSAGE);
          inputB = JOptionPane.showInputDialog(null, "Specify the Coefficient B: ", HEADING,  +
                  JOptionPane.QUESTION_MESSAGE);
         inputC = JOptionPane.showInputDialog(null, "Specify the Coefficient C: ", HEADING,  +
                  JOptionPane.QUESTION_MESSAGE);
         inputX = JOptionPane.showInputDialog(null, "Specify the x-value: ", HEADING,  +
                  JOptionPane.QUESTION_MESSAGE);

         // Convert the string inputs to numeric data:
         coeffA = Integer.parseInt(inputA);
         coeffB = Integer.parseInt(inputB);
         constC = Integer.parseInt(inputC);
         carX = Integer.parseInt(inputX);

          // Print equation and calculated y-value:
         //System.out.println("You have specified the parabola defined by the formula: ");
         //System.out.println("\t" + "y = " + coeffA + "x*x + " + coeffB + "x + " + constC);
         //System.out.println();
         outputString1 = "You have specified the parabola defined by the formula: " +
                  "\t" + "y = " + coeffA + "x*x + " + coeffB + "x + " + constC + "\n";

         varY = coeffA * varX * varX + coeffB * varX + constC;
         outputString2 = "The y-value for x-value of " + varX + " is " + varY;
         //System.out.println("The y-value for x-value of " + VarX + " is " + VarY);
         JOptionPane.showMessageDialog(null, outputString1 + outputString2,  +
                  HEADING,JOptionPane.INFORMATION_MESSAGE);

   } // End main Method
} // End of class Parabola
```

2.10 Getting Input From the Console

Earlier versions Java do not ship with methods for reading input from the console. However, you can write your own. Fortunately, several individuals have done this ahead of you (for instance, see [Liang 2015] or [Savitch 2014]). In Liang's **MyInput** class, the methods readByte(), readShort(), readInt(), readLog(), readFloat(), readDouble(), readChar(), readBoolean(), and readString() are defined. Each method returns data of the type read.

Current versions of Java provide various classes that you can use to obtain and process input from the keyboard. The **Scanner** class is one of them. We will not fully discuss this class here (it will be revisited later in the course). It is mentioned here just to explain how it can be used to get obtain input from the keyboard:

- In Java, the keyboard is denoted by **System.in**
- **Scanner** can be used to fetch inputs as an input stream (white space is used as the separator for various data elements in the input). The class contains various methods for extracting various data inputs (according to the primitive types) from the input stream.

Figure 2.21 lists the data extracting methods of the **Scanner** class, while figure 2.22 provides an illustration of how it has been used to develop a class called **EFInput3**. This class is then used to read input from the keyboard via its methods: **readString()**, **readInteger()**, **readShort()**, **readLong()**, **readFloat()**, **readDouble()**, **readChar()**, **readByte()**, and **readBoolean()**. To use **EFInput3** in your program to read a string from the keyboard, you would use the following construct:

```
<StringVar> EFInput3.readString();
// Example:
String myName = EFInput3.readString();
```

Figure 2.21: Data Extraction Methods of the Scanner Class

Scanner Method	Comment
String next()	Returns the next input as a string
String nextLine()	Returns all remaining inputs on the current line as a string.
boolean nextBoolean()	Returns the next input as a Boolean data item.
byte nextByte()	Returns the next input as a byte data item.
double nextDouble()	Returns the next input as a double data item.
float nextFloat()	Returns the next input as a floating point data item.
int nextInt()	Returns the next input as an integer data item.
long nextLong()	Returns the next input as a long integer data item.
short nextShort()	Returns the next input as a short integer data item.

Figure 2.22: Using Scanner Class to Retrieve Keyboard Input

```
/* EFInput3.java                          Author Elvis Foster                                    */
// ***************************************************************************
/* List of Methods:
 readString(), readInteger(), readShort(), readLong(), readFloat(), readDouble(), readChar(),
 readByte(), readBoolean()                                                                      */
// ***************************************************************************
package javaapplication2;
import java.util.*; // Facilitates use of Scanner class, StringTokenizer class, etc
import java.io.*; // Facilitates I/O to and from standard input
//import java.io.BufferedReader;
//import java.io.InputStreamReader;

public class EFInput3
{
  //Global Declaration(s)
  static Scanner ScanInput = new Scanner(System.in); // For obtaining input from the keyboard

  // readString Method
  public static String readString()throws Exception
  {
   String Result = null;
   final String ErrorMsg = "Fatal error during attempt to read a string from the keyboard: ";
   try
   {
     Result = ScanInput.next().trim();
     return Result; // Returns the next string from the (keyboard) input to the calling statement
   }
   catch (Exception Ex1)
   {
     // System.out.println(Ex.getMessage( ));
     Exception Ex2 = new Exception(ErrorMsg + Ex1.getMessage());
     throw Ex2;
      //    System.exit(0);
    }
 } // End readString
 // ...
  // readChar Method
  public static char readChar() throws Exception
  {
   char Result = '0';
   try
   {
     Result = ScanInput.next().charAt(0);
     return Result; // Returns the next character from the (keyboard) input to the calling statement
   }
   catch (Exception Ex)
   {
     throw Ex;
   }
  } // End readChar

} // End EFInput3
```

2.11 The String Class

You are already familiar with the type **String** and the concatenation operator (+). **String** is actually a class. Declaring a string variable actually creates an instance of the **String** class. This object therefore has access to all the member methods of the **String** class. Figure 2.23 provides *method signatures* (i.e. method headings) and explanations for some of the commonly used methods of the **Strin**g class, while figure 2.24 provides a basic illustration of how to use the methods to manipulate strings. The convention used in listing the methods (column 1 of figure 2.23) is to state the *return type*, followed by the name of the method. The return type is a valid primitive type or class that describes the value that a given method will return to its calling statement. You will learn more about methods in lecture 4.

Figure 2.23: Commonly Used Methods of the String Class

String Method Signature	Comment
char charAt(int index)	Return the character at the specified position in the string.
int compareTo(String OtherString)	Compares the calling string with **OtherString**. Returns –ve value if the calling string is first, zero if the strings are equal, and +ve if **OtherString** is first.
String concat(String OtherString)	Concatenates **OtherString** to the end of the calling string.
boolean equals(Object anObject)	Returns **true** if the calling string is equal to the other object (string) specified; **false** otherwise.
boolean equalsIgnoreCase(Object anObject)	Same as **equals(…)** except that the case is ignored.
int indexOf(int ch)	Returns the index of the first occurrence of specified character in the calling string.
int indexOf(int ch, int FromIndex)	Returns the index of the first occurrence of specified character in the calling string, starting at **FromIndex**.
int indexOf(String ThisString)	Returns the index of the first occurrence of specified string (**ThisString**) in the calling string.
int indexOf(String ThisString, int FromIndex)	Returns the index of the first occurrence of specified string (**ThisString**) in the calling string, starting at **FromIndex**.
int lastIindexOf(String ThisString)	Returns the index of the last occurrence of specified string (**ThisString**) in the calling string.
int lastIindexOf(String ThisString, int FromIndex)	Returns the index of the last occurrence of specified string (**ThisString**) in the calling string, starting at **FromIndex**.
int length()	Returns the length of the string.
String substring(int FromIndex)	Returns a substring, starting at **FromIndex** to the end of the calling string.
String substring(int FromIndex, int ToIndex)	Returns the substring between **FromIndex** and **ToIndex - 1** in the calling string.
String toLowerCase()	Returns the string converted to lower case.
String toUpperCase()	Returns the string converted to upper case.
String trim()	Returns the string stripped of all leading and trailing white space.
String toString()	Returns itself.

Java also has other string-related classes such as **StringBuffer**, **StringTokenizer**, and **Scanner**. You will learn more about these later in the course.

Figure 2.24: Illustrating String Manipulation

```
/* StringIllustration.java                                                    */
/* Created on June 23, 2005, 11:00 AM                                         */
/* Author Elvis Foster                                                        */
// ***************************************************************************
package javaapplication2;
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.

public class StringIllustration
{ // Beginning of StringIllustration
  /* No Constructor needed
   public StringIllustration() {} */

   public static void main(String[] args)
   { // Begin main method
   // Class Assignment #1: This program manipulates some strings
    String FirstName, MiddleName, LastName, Initials;
    final String HEADING = "String Illustration";

    // Accept the names of a visitor:
    FirstName = JOptionPane.showInputDialog(null, "Please enter your first name: ", HEADING,  +
        JOptionPane.QUESTION_MESSAGE);
    MiddleName = JOptionPane.showInputDialog(null, "Please enter your middle name: ", HEADING,  +
        JOptionPane.QUESTION_MESSAGE);
    LastName = JOptionPane.showInputDialog(null, "Please enter your last name: ", HEADING,  +
        JOptionPane.QUESTION_MESSAGE);


    // Determine ant print the initials of the visitor:
    Initials = (FirstName.substring(0, 1) + MiddleName.substring(0, 1) +
        LastName.substring(0,1)).toUpperCase();
    System.out.println("Your initials are " + Initials);
    JOptionPane.showMessageDialog(null, "Your initials are " + Initials,  +
        HEADING,JOptionPane.INFORMATION_MESSAGE);
   } // End main method

} // End of StringIllustration
```

2.12 The Character Class

As mentioned earlier, Java provides a class for each primitive type; these classes are stored in the package called **java.lang**. Like strings, quite often, you will need to manipulate characters. The **Character** class is precisely for that purpose; it provides you with a number of methods to do just that. Figure 2.25 provides a method signatures and explanations for the commonly used **Character** methods.

Figure 2.25: Commonly Used Methods of the Character Class

Character Method Signature	Comment
static int digit(char ch, int radix)	Returns the numeric value of the character ch in the specified radix.
boolean equals(Object anObject)	Returns **true** if the calling string is equal to the other object (string) specified; **false** otherwise.
static boolean isDigit(char ch)	Returns **true** if the specified character is a digit; **false** otherwise.
static boolean isLetter(char ch)	Returns **true** if the specified character is a letter; **false** otherwise.
static boolean isLetterOrDigit(char ch)	Returns **true** if the specified character is a letter or a digit; **false** otherwise.
static boolean isLowerCase(char ch)	Returns **true** if the specified character is in lower case; **false** otherwise.
static boolean isUpperCase(char ch)	Returns **true** if the specified character is in upper case; **false** otherwise.
static char toLowerCase(char ch)	Returns the specified character converted to lower case.
static char toUpperCase (char ch)	Returns the specified character converted to upper case.
Note: Static methods can be called directly, without an instance of the class being created.	

2.13 Formatted Output

You are already familiar with **System.out.print(…)** and **System.out.println(…)**. Java also provides a **System.out.printf(…)** method, which allows for formatted output. In order to use it, you need to include at least one format specifier in your output string. For each format specifier, you must include a variable or expression that provides a value matching that specifier. The variable(s) and/or expression(s) are to be stated positionally, matching the order in which the specifiers have been stated. This is best explained with an example:

Example 10: The code segment below illustrates the use of format specifiers.

```
float hoursWorked, wage;
…
System.out.printf("Hours worked is %f5.2 and your wage is %f7.2", hoursWorked, wage);
```

In this example, the output includes two floating point values: the first corresponds to the variable **hoursWorked**, and is length five characters with two decimal places; the second corresponds to variable **wage**, and is of length seven characters with two decimal places.

2.13 Formatted Output (continued)

Figure 2.26 provides a list of valid specifiers. Note that for specifiers **d**, **f**, **e**, and **s**, you can specify the desired lengths as in the example above.

Figure 2.26: Frequently Used Specifiers

Specifier	Output	Example
%b	A Boolean value	**true** or **false**
%c	A character	'E'
%d	A decimal value	2001
%f	A floating point number	56.876
%e	A number in scientific notation	6.448401E + 05
%s	A string	"Java is awesome"
Note: The % sign denotes a specifier. To output a literal %, in the format string, use %%.		

There is a much slicker way to format numerical data for output, but it involves the use of more advanced feature of Java — the **DecimalFormat** class. We will defer a discussion of this until lecture 4.

2.14 Keeping Track of Date and Time

The are two commonly used approaches for keeping track of date and time in Java. One involves the use of the method **System.currentTimeMillis()**; the other involves the **Calendar** and **GregorianCalendar** classes.

Using System.currentTimeMillis():

The **System.currentTimeMillis()** method stores time in milliseconds since January 1, 1970 (this date is associated with the introduction of the Unix operating system). This method can therefore be used to calculate the current time as follow (figure 2.27 illustrates):

- Obtain Milliseconds.
- Divide **Milliseconds** by 1000 to obtain **TotalSeconds**.
- Compute **CurrentSecond** in the minute in the hour by determining **TotalSeconds** % 60.
- Computer the **TotalMinutes** by dividing **TotalSeconds** by 60.
- Compute the **CurrentMinute** in hour from **TotalMinutes** % 60.
- Compute **TotalHours** by dividing **TotalMinutes** by 60.
- Compute **CurrentHour** by determining **TotalHours** % 24.

Using the Calendar Class:

A more elegant approach is to use the Calendar class. However, this requires a deeper understanding of and familiarity with inheritance. A full discussion is therefore deferred until later in the course (lecture 6). However, a program listing, depicting the approach is provided in figure 2.28 for the curious mind.

Figure 2.27: Using System.currentTimeMillis() to Determine Date & Time

```
/* EFDate0.java                                                                    */
/* Created on February 23, 2005, 6:50 PM                                          */
/* Illustrating the date manipulation                                             */
/* Author Elvis Foster                                                            */

package javaapplication2;
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.

public class EFDate0
{
  // Constructor not required here
  public EFDate0()  {  }

  public static void main(String[] args)
  { // Start main
   long totalMilliseconds = System.currentTimeMillis(); // Obtain total milliseconds since Jan 1, 1970
   long totalSeconds = totalMilliseconds / 1000; // Compute total seconds since Jan 1, 1970
   int currentSecond = (int) (totalSeconds % 60); // Compute current second in the minute in the hour
   long totalMinutes = totalSeconds / 60; // Compute total minutes
   int currentMinute = (int) totalMinutes %60; // Compute current minute in the hour
   long totalHours = totalMinutes / 60; // compute total hours
   int currentHour = (int) (totalHours % 24); // Compute current hour

   String CurrentTime =  "The current time is " + currentHour + ": " + currentMinute + ": " +
         currentSecond + " GMT";
   JOptionPane.showMessageDialog(null, CurrentTime, "Date Illustration", +
         JOptionPane.INFORMATION_MESSAGE);

  } // End main
} // End EFDate0
```

Figure 2.28: Using the GregorianCalendar Class to Determine Date and Time

```
/* EFDate.java: Illustrating the date manipulation                                    */
/* Author Elvis Foster                                                                */
// ***********************************************************************************

package javaapplication2;
import java.util.*; // Facilitates use of the Calendar class
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.

public class EFDate
{
  // Constructor not required here
  public EFDate() {  }

  public static void main(String[] args)
  {
    //
    Calendar CurrentDate = new GregorianCalendar(); // Date will be in Gregorian format
    String OutputString;
    int currentMonth;
    // Format the date, then print it.
    currentMonth = CurrentDate.get(Calendar.MONTH) + 1;
    OutputString = "The current date is: " + currentMonth + "-" +
          CurrentDate.get(Calendar.DAY_OF_MONTH) + "-" + CurrentDate.get(Calendar.YEAR) +
          "\n" + "Day of the week is " + CurrentDate.get(Calendar.DAY_OF_WEEK) + "\n" +
          "Week of the year is " + CurrentDate.get(Calendar.WEEK_OF_YEAR) + "\n" +
          "Day of the year is " + CurrentDate.get(Calendar.DAY_OF_YEAR) + "\n" +
          "The time is " + CurrentDate.get(Calendar.HOUR_OF_DAY) + ":" + CurrentDate.get(Calendar.MINUTE) + ":" +
          CurrentDate.get(Calendar.SECOND);

    JOptionPane.showMessageDialog(null, OutputString, "Date Illustration",  +
          JOptionPane.INFORMATION_MESSAGE);

  } // End main method
} // End EFDate class
```

2.15 Java Keywords

Every programming language has a set of reserve words or keywords which you are forbidden to us for any other purpose. The Java keywords are listed in figure 2.29.

Figure 2.29: Java Keywords

Primitive Types	
byte boolean char double float int long short void	
Logic Control Keywords	
break	Used in switch-statement or in loops
case	Defines a set of statements to be executed if a value specified matches the value defined by an enclosing **switch** keyword
continue	Used in loops
default	Optionally used after the last case statement of a switch statement. If no case conditions are matched, the default statement will be executed.
do	Used in do-while statement
else	Used in if-else-statement
false	Boolean value
for	Used in for-statement
if	Used in if-statement
switch	Defines a values that an expression may have, and action to be taken for each possible value.
true	Boolean value
while	Used in while-statement
Declaration Keywords	
abstract	Used for abstract classes & methods
class	Used for class definition
extends	Used for class inheritance
final	Used for constant declaration
interface	Used for interface definition
implements	Used for interface inheritance
package	Used in defining a package or linking a class to a package
public	Public resources (data item, class or method)
private	Public resources (data item, class or method)
protected	Protected resources (data item, class or method)
main	Used for the **main** method
static	Used to declare static resources (data items or methods)
strictfp	Introduced in Java 1.2 to ensure that calculations were always strict floating point, meaning **float** or **double.** Included in current versions for backward compatibility.
Other Keywords	
assert	Used in exception handling
import	Used to import Java resources in various packages
instanceof	Special operator used in checking the rightful owner of an object
catch	Used in exception handling
finally	Used in exception handling
new	Special operator to create an instance of a class
null	Null value
return	Used to return a value from a method
super	Refers to the parent of a class instance
this	Refers to the current instance of a class
throw	Used in exception handling
throws	Used in exception handling
try	Used in exception handling
const	C++ keyword that Java accepts
goto	C++ keyword that Java accepts
native	
synchronized	
transient	
volatile	

2.16 Commonly Used Java Packages

If you go to the Oracle Java Documentation site [Oracle 2014], one of the first things you will notice is that a Java development environment includes several packages. It is unlikely that you will be familiar with all of these packages; however, you should be cognizant of the commonly used ones. Figure 2.30 provides a list of them. As you go through this course, these will be introduced to you, so there is no need to be overwhelmed by them.

Figure 2.30: Commonly Used Java Packages

Java Package	Comment
Java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
Java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
Java.beans	Contains classes related to developing *beans* -- components based on the JavaBeans architecture.
Java.io	Contains classes for system input and output through data streams, serialization and the file system.
Java.lang	Provides classes that are fundamental to the design of the Java programming language.
Java.security	Provides the classes and interfaces for the security framework.
Java.sql	Provides the API for accessing and processing data stored in a data sources (usually a relational databases) using the Java programming language.
Java.text	Provides classes and interfaces for handling text, dates, numbers, and messages in a manner independent of natural languages.
Java.util	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes.
Java.crypto	Provides the classes and interfaces for cryptographic operations.
Java.management	Provides the core classes for the Java Management Extensions.
Java.naming	Provides the classes and interfaces for accessing naming services.
Java.net	Provides classes for networking applications.
Javax.secutity.auth	This package provides a framework for authentication and authorization.
Javax.sql	Provides the API for server side data source access and processing from the Java™ programming language.
Javax.swing	Provides a set of "lightweight" GUI components that, to the maximum degree possible, work the same on all platforms.
Javax.xml	Defines core XML constants and functionality from the XML specifications.

2.17 Summary and Concluding Remarks

We have covered a lot of Java fundamentals in this chapter. It is important that you grasp these concepts and principles. Here is a brief summary of the salient points covered:

Java is an OOPL that provides a number of distinguishing features such as:

- The JVM which facilitates platform independence;
- Relative simplicity, compared to C++, the language on which it was modeled;
- Wide usage in network programming as well as internet programming;
- Support for multithreading;
- Contains built-in features that support GUI development;
- Typically marketed in an interpretive environment.

2.17 Summary and Concluding Remarks (continued)

The anatomy of a Java program may be summarized as follows:
- A Java program is essentially made up of one or more classes.
- Classes are made up of one or more methods and data items.
- Methods are made up of instructions (in the form of statements) and possibly data items.
- Applications are made up of one or more classes; an application is implemented as a package.

Two easy ways to send output to the computer monitor are via the **System.out.print(…)** method, or **JOptionPane·showMessageDialog(…)** method.

Primitive data types in Java are byte, short, int, long, float, double, char, boolean, void, and enum.
Commonly used Java qualifiers are public, private, protected, static, final, and abstract.
Typical operators for arithmetic expressions: + - * / % ++ -- =
Shortcut arithmetic operators: += -= *= /= %=
Typical Boolean operators: < <= == != > >= || && ! & | ?

In expressions containing data items of dissimilar data types, automatic conversion takes place when necessary, from the lower range (e.g. int) to the higher range (e.g. float). Whenever it is necessary to convert from a higher ranger to a lower range, or across different data types, an explicit casting is required. Data conversion does not affect the originally declared data items (variables), only the instance(s) occurring in that specific expression.

In designing programs using an OOP, it is important to be comfortable with the following concepts:
- An object type is a concept or thing about which data is stored.
- An operation is a task that can be performed on an object.
- A method is a set of instructions for carrying out an operation
- When designing software systems, it is desirable to talk about object types and operations; at the implementation level when the program is being designed and written, it is more desirable to talk about classes and methods. Classes implement object types and methods implement operations.
- The structure of a class refers to the data items and methods that have been defined within the class. The data items and methods are also referred to as the properties of the class.
- Amalgamation refers to the case where an object is comprised of other objects. The amalgamation may be an aggregation or a composition.
- An object inherits all the properties of its parent class**.** Additionally, a class may inherit from another class. The inheriting class is referred to as the sub-class and the inherited class is called the super-class.
- Polymorphism is the act of an object or method taking on different forms depending on the circumstance.

One way to obtain input from the keyboard are via the Java **JOptionPane.showInputDialog(…)** method. When using this method, it is important to remember that only strings are accepted. Conversion to other desired primitive data types can be achieved by remembering that for each primitive data type, Java provides a class which contains at least one data-conversion method to convert a string to the primitive data type represented by that class. For example, to convert from string to float, use the **parseFloat(…)** method in the **Float** class.

2.17 Summary and Concluding Remarks (continued)

Another way to obtain input from the keyboard via the Java **Scanner** class. This class contains various methods to read data of the various primitive data type.

The **String** class contains various methods for manipulating strings. For instance, you may concatenate a string to another, trim trailing and/or leading blanks from a string, convert a string to upper or lower case, extract a substring from the string, etc. Similarly, the **Character** class allows for the manipulation of characters.

You may keep track of time by using the **System.currentTimeMillis()** method. However, a much clicker way is to make use of the **Calendar** class and its various sub-classes; the **GregorianCalendar** class is particularly useful.

The Java language has a number of keywords and reserve-words; they may not be used for any other purpose than their original intent.

Java also ships with a number of built-in packages, each containing various classes. As you learn the language, you will encounter some of these packages.

2.18 Review Questions

1. What are some of the salient features of the Java programming language?

2. Briefly explain the Java compilation process.

3. Outline and clarify the anatomy of a Java program.

4. Practice writing simple Java programs to output information on the screen, using the **System.out.print(…)** method and the **JOptionPane·showMessageDialog(…)** method. Explain the difference between the two. Which do you prefer?

5. What are the primitive data types in Java?

6. Practice declaring and manipulating variables of different primitive data types. Your manipulations should include using the various arithmetic operators, the assignment statement, and outputting information on the screen.

7. Practice writing expressions that include the Java escape characters, and observe their effects.

8. Practice writing expressions that involve data conversion — both automatic promotion and explicit casting.

9. Make a rough mental note of the Java operator precedence table. As your knowledge of Java expands, you will be using more and more of these operators.

2.18 Review Questions (continued)

10. Explain with appropriate examples, the following concepts: object type, operation, class, method, structure, inheritance, polymorphism, and amalgamation.

11. Practice writing simple Java programs that obtain input from the keyboard via the **JOptionPane.showInputDialog(…)** method, manipulate related variables, and output information to the screen.

12. Practice writing simple Java programs that obtain input from the keyboard via the **Scanner** class instead of the **JOptionPane.showInputDialog(…)** method, manipulate related variables, and output information to the screen.

13. Write a simple program that prompts the user for various strings (three or four), performs various manipulations of them, and then produces some meaningful output.

14. Write a simple program to retrieve the date and display it on the screen.

15. Observe the list of Java keywords provided.

16. Observe the list of commonly used Java packages that has been provided.

2.19 Recommended Readings

[Bell & Parr 2010] Bell, Douglas and Mike Parr. 2010. *Java for Students* 6th Ed. New York: Pearson. See chapters 1 – 4.

[Liang 2014] Liang, Y. Daniel. 2014. *Introduction to Java Programming — Comprehensive Version*, 10h ed. Boston, MA: Pearson Education. See chapters 1, 2, and 4.

[Savitch & Carrano 2008] Savitch, Walter and Frank M. Carrano. 2008. *Java: An Introduction to Problem Solving & Programming* 5th ed. Upper Saddle River, NJ: Prentice Hall. See chapters 1and 2.

[Oracle 2015]. Oracle Corporation. 2015. "Java™ Platform, Standard Edition 8API Specification." Accessed January 19, 2015. http://docs.oracle.com/javase/8/docs/api/