Essential Elements of Java

Objectives

- Understand some basic Java language elements.
- Be able to identify different types of errors in a program.
- Know how to use simple output operations.
Outline

- Programming errors
- Java Language Elements
  - program structure
  - identifiers and keywords
  - data types and variables
  - statements
  - messages
  - class definitions
- The System and PrintStream classes.

Syntax

- The syntax of a programming language is the set of rules that determines whether its statements are correctly formulated.
- Example Rule: All Java statements must end with a semicolon.
- Syntax error: question = q
- Syntax errors can be detected by the compiler, which will report an error message.
Semantics

- The *semantics* of a programming language is the set of rules that determine the meaning of its statements.
- Example Rule: In $a + b$, the $+$ operator will add $a$ and $b$.
- *Semantic error:* User intended to add $a$ and $b$ but coded $a - b$.
- Semantic errors cannot be detected by the compiler, because it can’t read the programmer’s mind.

Errors

- A *syntax error* results when a statement violates one of Java’s grammar rules. The java compiler catches *syntactic errors*, producing error messages.
- A *semantic error* or *logic error* is an error in the program’s design and **cannot be detected by the compiler**.
- Semantic errors produce incorrect output or behavior when the program is running.
Testing and Debugging

- The programmer must test thoroughly for semantic errors.
- Testing a program can only reveal the presence of bugs, not their absence.
- It is takes less time to avoid errors through slow and careful coding than to detect and fix them later.

Writing Readable Programs

- Style, in addition to working code, is the mark of a good programmer. Style consists of:
  - Readability.
    - Code should be well-documented and easy to understand.
  - Clarity.
    - Conventions should be followed and convoluted code avoided.
  - Flexibility.
    - Code should be designed for easy maintenance and change.
**Java Program Structure**

- A Java program is made up of *class definitions*.
- Each public class definition is written in a separate *source code* file with the same name as the class plus the `.java` extension.
- The Java compiler produces *bytecode* files with the same names as the source files but with a `.class` extension.
- A class definition contains a *header* and a *body*.
- The body contains *field* and *method* definitions.
- One of the classes in a program must contain a main method, which is where the program execution begins.

**Java Language Elements: Identifiers**

*(names of classes, fields, methods, etc.)*

- An *identifier* must begin with a letter (A to Z, a to z) and may be followed by any number of letters or digits (0 to 9) or underscores (_).
- An identifier may not be identical to a Java keyword.
- **Style:** Class names begin with a capital letter and use capitals to distinguish words within the name. e.g: HelloWorld, TextField
- **Style:** Variable and method names begin with a lowercase letter and use capitals to distinguish words within the name, e.g: main(), canFly()
Java Keywords

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

Data Types

- Java data are classified according to data type.
- Object types (Ball, Penguin, String) versus primitive types (int, double, boolean).

<table>
<thead>
<tr>
<th>Kind</th>
<th>Type (keyword)</th>
<th>Size in Bits</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>-</td>
<td>true or false</td>
</tr>
<tr>
<td>character</td>
<td>char</td>
<td>16</td>
<td>‘A’, ‘5’, ‘+’</td>
</tr>
<tr>
<td>integer</td>
<td>byte</td>
<td>8</td>
<td>-128 to +127</td>
</tr>
<tr>
<td>integer</td>
<td>short</td>
<td>16</td>
<td>-32768 to +32767</td>
</tr>
<tr>
<td>integer</td>
<td>int</td>
<td>32</td>
<td>-2147483648 to +2147483647</td>
</tr>
<tr>
<td>integer</td>
<td>long</td>
<td>64</td>
<td>really big numbers</td>
</tr>
<tr>
<td>real number</td>
<td>float</td>
<td>32</td>
<td>21.3, -0.45, 1.67e28</td>
</tr>
<tr>
<td>real number</td>
<td>double</td>
<td>64</td>
<td>21.3, -0.45, 1.67e28</td>
</tr>
</tbody>
</table>

Java’s Primitive Data Types.
**Variables**

- A *variable* is a container that can store a value of a certain type. (A field is a special kind of variable that is used to store the value of an object’s attribute.)

![Diagram of a variable](attachment:variable_diagram.png)

**Figure 1.6. A variable is a typed container.**

**Fields versus Local Variables**

- Fields are declared in the body of a class definition outside of any method.
- Fields are used to maintain information about an object’s state. The storage space for a field is reserved for as long as the object exists.
- Local variables are declared inside a method.
- Local variables are used as temporary scratch space to store intermediate results during the execution of a method. The storage space is only reserved until the method execution is complete.
Statements

- A *statement* is a segment of code that produces some action in a program.
- Statements are terminated by semicolons.
- A *compound statement* consists of a sequence of other statements inside curly brackets.
- A *block* is a compound statement that includes declaration statements.

Declaration Statements

- A *declaration statement* creates a variable (storage container). It does not create an object to put in the container.
- A declaration has the type followed by the variable name:
  ```
  int number;
  Penguin fred;
  ```
- Multiple variables of the same type can be declared in a single statement:
  ```
  int i, j, k;
  Ball b1, b2;
  ```
- Declaration statements can include *initializers*:
  ```
  int i = 5;
  Ball b1 = new Ball(Color.yellow, 10);
  ```
Assignment Statements

- An assignment statement stores a value in an existing variable, possibly replacing a previous value. It does not create a new storage container.
- An assignment statement has a variable name followed by an equal sign and the value to assign. It does not have a type name:
  
  ```
  i = 10;
  b1 = new Ball(Color.green, 20);
  ```

  ![caution]
  It is possible to create more than one variable with the same name. Do not write a declaration statement (by using a type name) when you intend to assign a new value to an existing variable or field with an assignment statement!

Expressions and Operators

- An expression is Java code that specifies or produces a value in the program.
- Expressions may use literals, variables, and operators (+, -, <, >, …)

  ```
  i + j    // An addition of type int
  i < j    // A comparison of type boolean
  num == 7  // An equality comparison of type boolean
  ```

- Expressions occur within statements:

  ```
  num = i + j; // An assignment statement
  ```
Strings

- A Java *string* is an object that represents a sequence of characters.
- Strings are the only **objects** in java that can be specified using literal values, e.g. “Hello”.
- String objects can be created without using the new operator:
  ```java
  String s = new String(“Hello”);
  or:  String s = “Hello”;
  ```

String Methods

```java
String s1 = “Hello”;  
String s2 = “ World!”; 
String s3 = “Hello”; 
int n = s1.length();  // n = 5 
boolean b1 = s1.equals(s2);  // b1 = false 
boolean b2 = s1.equals(s3);  // b2 = true 
String s4 = s1.concat(s2);  // s4 = “Hello World!”; 
String s5 = s1 + s2;  // s5 = “Hello World!”;
```
Messages

- A message starts with the variable name of the object to receive the message, then a dot followed by the name of the message and a list of argument values in parentheses:
  ```
  b1.move(10, 20);
  ```
- The parentheses are always included, even if there are no arguments.
- Arguments don’t have to be literals. They can be specified by any expression that produces a value of the correct type.
- A message that produces a return value can be used in an expression.

Keyword this

- An object may refer to itself using the keyword `this`.
- `this` is most commonly used when an object sends itself a message (to invoke one of its own methods).
- Suppose we wanted Balls to redraw themselves in the proper location whenever they moved:

```java
public void moveToRandomPosition(Window w) {
    Graphics g = w.getGraphics();
    this.erase(g);
    centerX = ...
    centerY = ...
    this.draw(g);
}
```
Method Signatures

- The object receiving a message executes its corresponding method in response.
- In determining which method to use in response to a message, the method signatures are considered, not just the names.
- A method signature consists of the name plus the types of parameters (but not the return type).
- For example, the `drawString` method in the `Graphics` class has the signature:
  ```java
drawString(String, int, int)
```

Overloading

- It is OK for two methods of a class to have the same names as long as the signatures are different. This is called overloading.
- The `println` method in the `PrintStream` class is heavily overloaded. There are many different methods with the name `println` but with different parameter types.
- Don’t confuse overloading with overriding.
  - Overriding replaces an inherited method with a new method which has the same signature.
  - Overloading adds another method with the same name but different signature than some other method in the class (inherited or not).
Class Definition

- A Java program consists of one or more class definitions.
- A class definition contains a class header:
  \[
  \text{ClassModifiers}_{\text{opt}} \ \text{class} \ \text{ClassName} \ \text{Pedigree}_{\text{opt}}
  \]
  
  ```java
  public class HelloWorld extends Object
  ```

- And a class body, which is code contained within curly brackets: {...} 

  ```java
  public class HelloWorld extends Object
  {
  }
  ```

Java Language Elements:
Class Definition

```java
public class HelloWorld extends Object // Class header
{
    // Start of class body
    private String greeting = "Hello World!"; // Field definition

    public void greet() // Method definition header
    {
        // Start of method body
        System.out.println(greeting); // Output statement
    } // End of greet method body

    public static void main(String[] args) // Method definition header
    {
        // Start of method body
        HelloWorld helloworld; // Local variable
        helloworld = new HelloWorld(); // Object instantiation
        helloworld.greet(); // Message
    } // End of method body
} // End of class body
```
Declaring a Field

- In general a field declaration takes the following form:
  \[ \text{Modifiers}_{\text{opt}} \text{ Type FieldName InitializerExpression}_{\text{opt}} \]

  ```java
  private String greeting = "Hello World";
  private int num;
  private double realNum = 5.05;
  private int count = 0;
  ```

  In these examples the types are String, int, double and the field names are greeting, num, realNum, and count.

- **Remember**: The class does not contain any field values (even when an initializer is given). Only the objects have field values!

Declaring a Method

- A method definition consists of a method header:
  \[ \text{Modifiers}_{\text{opt}} \text{ ReturnType MethodName (ParameterList}_{\text{opt}} \]

- Followed by a method body, which is executable code contained within curly brackets: `{...}`

  ```java
  public void greet() // Method definition header
  { // Start of method body
    System.out.println(greeting); // Executable statement
  } // End of greet method body
  ```

  ```java
  public static void main(String[] args) // Method definition header
  {
    HelloWorld helloworld; // Local variable
    helloworld = new HelloWorld(); // Object creation/assignment
    helloworld.greet(); // Message to object
  }
  ```
Java Library: System and PrintStream

- The java.lang.System class contains PrintStream objects that perform Input/Output (I/O).

<table>
<thead>
<tr>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ out : PrintStream</td>
</tr>
<tr>
<td>+ err : PrintStream</td>
</tr>
<tr>
<td>+ in : InputStream</td>
</tr>
</tbody>
</table>

The java.lang.PrintStream class contains the print() and println() methods that perform output.

<table>
<thead>
<tr>
<th>PrintStream</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ print(in data : String)</td>
</tr>
<tr>
<td>+ print(in data : boolean)</td>
</tr>
<tr>
<td>+ print(in data : int)</td>
</tr>
<tr>
<td>+ println(in data : String)</td>
</tr>
<tr>
<td>+ println(in data : boolean)</td>
</tr>
<tr>
<td>+ println(in data : int)</td>
</tr>
</tbody>
</table>

Example: OldMacDonald Program

```java
public class OldMacDonald {
    public static void main(String[] args) {
        String hadAFarm = "Old MacDonald had a farm."
        String onHisFarm = "And on his farm he had a ");

        System.out.println(hadAFarm);
        System.out.println("E I E I O");
        System.out.print(onHisFarm);
        System.out.println("duck");
        System.out.println("E I E I O");
        System.out.println("With a quack quack here.");
        System.out.println("And a quack quack there.");
        System.out.println("Here a quack, there a quack.");
        System.out.println("Everywhere a quack quack.");
        System.out.println(hadAFarm);
        System.out.println("E I E I O");
    }
}
```
Technical Terms

- assignment statement
- block
- comment
- compound statement
- data type
- declaration statement
- expression
- identifier
- keyword
- literal value
- object instantiation
- operator
- parameter
- primitive data type
- semantics
- syntax