

Java information sheet

Syntax vs. Semantics vs. Style

- A syntax rule is a requirement imposed by the Java programming language; e.g., a variable must be defined before it can be used.
- A semantics rule defines how something works; e.g., in Java the + operator performs addition when it has numeric operands.
- A style rule is a programming convention to program appearance; e.g., a Java constant should only use uppercase letters and underscores in its name.

Comments

- The sequence // indicates the rest of the line is not to be translated as part of the program. The text is a comment to a program reader.
- Programs often have header comment section that identifies the purpose of the program, the authors, and contact information.
- A method normally begins with a comment indicating its purpose.
- A section of code within a method is typically preceded by a comment indicating its task.

Whitespace

- Whitespace between programming elements is ignored during compilation.
- Whitespace is used to separate program elements for increased readability.
- Indentation is used to indicate whether a program element is part of another program element; e.g., the statements within a method are indented with respect to the method.
- Standard indentation level is 3 or 4 spaces.
- If necessary for readability, long statements are normally broke up into multiple lines. The additional lines are indented with respect to the starting line of the statement.
- If necessary for readability, expressions long in length are broken immediately before an operator. If feasible, indenting the operator so that it lies below its operand is a standard practice.

Programs

- A Java program must have the name of its class be identical to the name file containing its definition.

- A Java class is a program if it has a method `public static void main()` taking a single parameter of type `String[]`. By convention the name of the parameter is `args`.

Keywords

- A keyword is a word reserved by Java and has special meaning within a program.
- All Java keywords are case sensitive and lowercase.
- The most frequently used keywords are `public`, `class`, `static`, `void`, `int`, `double`, `char`, `boolean`, `if`, `else`, `final`, `null`, `switch`, `while`, `for`, `throws`, and `catch`.

Identifiers

- Identifier is computing term for a name.
- A Java identifier must begin with a letter. Subsequent characters – if any – can be letters or numbers. Because Java uses the Unicode alphabet, letters from other languages are okay to use. The `_` and `$` are considered letters.
- Identifier names are case dependent; e.g., `peasPerPod` and `peasperpod` are different identifiers.
- Keywords cannot be used as names.
- The preferred naming scheme for a class is a noun whose letters are all lowercase except for the first letter of each word in the name; e.g., `DataCoordinate` rather than `datacoordinate`.
- The preferred naming scheme for a variable is all lowercase letters except for the first letter of each word after the initial word in the name; e.g., `peasPerPod` rather than `peasperpod`.
- The preferred naming scheme for a constant is all uppercase letters except for underscore characters separating the words within the name; e.g., `HOURS_PER_DAY` rather than `hoursPerDay` or `HOURSperDAY`.

Variables

- A variable is a symbolic name for a memory location whose contents can be modified during the execution of a program.
- A variable must be defined before it can be used.
- A variable must be initialized before it can be evaluated; that is, used in a calculation.
- The value of a primitive variable is a primitive value; i.e., a numeric or Boolean value.
- The value of an object variable is a reference – think pointer – to an object.

- A variable only exist within the block of statements containing its definition.
- A variable can be defined only once within a block of statements.

Constants

- A constant is a symbolic name for a memory location whose contents once they are initialized cannot be modified during the execution of a program.
- A constant must be defined before it can be used.
- A constant must be initialized before it can be evaluated; that is, used in a calculation.

Escape sequence

- An escape sequence allows a special character to be represented as a normal character.
- The primary escape sequences are
 - `\t` represents the tab character.
 - `\n` represents the newline character.
 - `\\` represents the backslash character.
 - `\'` represents the single quote character.
 - `\"` represents the double quote character.

Type

- A type is a collection of values along with the operators and methods that manipulate the values.
- Java divides its type into categories – primitive and object.
- The integer primitive types are **byte**, **char**, **short**, **int**, and **long**. These types respectively use 1, 2, 2, 4, and 8 bytes to represent an integer. **char** is also the character type.
- The floating point (decimal) primitive types are **float** and **double**. They respectively use 4 and 8 bytes to represent a decimal.
- An object typically has both attributes and behaviors.

Values and objects

- A literal value is either explicit primitive value or a character string; e.g., `19`, `2.8`, `true`, and `"aardvark"`.
- The value of a primitive type variable or constant is a value from its type.
- The value of an object variable is a reference to an object.
- Java uses keyword `null` to indicate a nonreference.
- There is a difference between uninitialized and `null`. The former indicates no value as of yet; the latter indicates the nonreference value.

- The dot operator `.` is the Java selection operator giving access to an element of an object; e.g., `s.x` is the `x` attribute of the object referenced by `s` and `s.f()` is the `f()` behavior of the object referenced by `s`.

System.out

- `System.out` represents the standard output stream; i.e.; where output should go by default.
- The `System.out.print()` method displays the value of its parameter to standard output.
- The `System.out.println()` method displays value of its parameter to standard output along with a newline character.
- When requesting input from a user it is standard to display a prompting message using the `System.out.print()` method.

Operators and evaluation

- When even close to doubt uses parentheses.
- Java provides the standard arithmetic operators `+`, `-`, `*`, and `/` along with the remainder (modulus) operator `%`.
- Java does not provide an exponentiation operator.
- The standard Java library defines a method `Math.pow()`. The method takes to double parameters. The method returns the value of the first parameter to the power of the second parameter.
- If the operands to an arithmetic operator are integer then, integer arithmetic is performed; e.g., `7 / 4` evaluates to the `int` value `1`.
- If the operands to a arithmetic operator are both numeric and at least one of the operands is a floating point value (decimal), then floating point arithmetic is performed; e.g. `7 / 4.0` evaluates to the `double` value `1.75`.
- The `+` operator performs concatenation when at least one of its operands is a `String` value; e.g., `"house" + "boat"` evaluates to the `String` `"houseboat"`; `"1" + "2"` evaluates to the `String` `"12"`; and `"1" + 2` evaluates to the `String` `"12"`.
- In expressions composed of more than one operator, Java uses precedence and associativity rules for determining the order of operator evaluation.
- The grouping operator `()` has higher precedence than unary, numeric, relational, logical, and assignment operators.
- The unary operators `+`, `-`, `!` have higher precedence than numeric, relational, logical, and assignment operators.
- The numeric operators have precedence than relational, logical, and assignment operators.

- The multiplicative operators `*`, `/`, and `%` have higher precedence than additive operators `+`, and `-`.
- The relational operators `<`, `<=`, `>`, and `>=`, `!=`, and `==`, have higher precedence than logical and assignment operators.
- The ordering operators `<`, `<=`, `>`, and `>=` have higher precedence than equality operators `!=` and `==`.
- Logical operators `&&` and `||` have higher precedence than the assignment operators.
- Operator `&&` has higher precedence than operator `||`.
- When operators have equal precedence, associativity rules determine the order of evaluation.
- The unary and assignment operators are evaluated right to left. Other operators are evaluated left to right.
- If the left operand of the `&&` operator is false, the right operand is not evaluated – the operation must be false.
- If the left operand of the `||` operator is true, the right operand is not evaluated – the operation must be true.
- The standard Java library `Math` provides methods for computing max, min, power, exponential, logarithm, and trigonometric functions.
- Standard library method `Math.pow()` performs exponentiation. The method takes two `double` parameters. The method returns the value of the first parameter to the power of the second parameter.

Assignment

- The assignment operator `=` replaces the value of its left operand with the value of its right operand.
- In an assignment, the left operand is called the target.
- The operator `++` increments the value of its operand by one; e.g., `++n` adds one to the value of variable `n`.
- The operator `--` decrements the value of its operand by one.
- The compound operators `+=`, `-=`, `*=`, `/=` and `%=` respectively increment, decrement, scale, divide, and modulate the target by the value of the right operand; e.g., `n += 5` increments the value of variable `n` by 5.

Strings

- Java uses class `String` for representing character strings.
- The `'x'` and `"x"` are different. The former is a `char`; the latter is a `String`.
- The `null` and `""` are different. The former indicates a nonreference; the latter is a `String` with length 0.
- Class `String` is part of the standard Java library `java.lang`.

- A character string within double quotes is a `String` literal; e.g., `"starting"`.
- `"name"` and `name` are different. The former is a `String` literal; the latter is an identifier.
- The characters in a `String` are accessible by their index. The first character has index 0, the second character has index 1, and so on.
- Suppose `s` is a `String`, `t` is a `String`, `u` is a `String`, `m` is an `int`, and `n` is an `int`.
 - `s.charAt(m)` is the `char` value of the character in `s` at index `m`.
 - `s.length()` is the number of characters in the character string represented by `s`.
 - `s.substring(m)` is a new string representing the substring of `s` starting with its character at index `m` and continuing to the end of `s`.
 - `s.substring(m, n)` is a new string representing the substring of `s` starting with its character at index `m` and continuing to the character with index `n-1`.
 - `s.indexOf(t, m)` is the `int` value equal to the index of the first occurrence of `t`'s character string within `s`'s character string when starting the search at index `m`. If there is no occurrence then the value of the method is `-1`.
 - `s.indexOf(t)` is equivalent to `s.indexOf(t, 0)`.
 - `s.lastIndexOf(t, m)` is the `int` value equal to the index of the last occurrence of `t`'s character string within `s`'s character string that starts on or before index `m`. If there is no occurrence then the value of the method is `-1`.
 - `s.lastIndexOf(t)` is equivalent to `s.lastIndexOf(t, m)`, where `m+1` is the length of character string represented by `s`,
 - `s.toLowerCase()` is a new string whose characters are the lowercase equivalents of the character string represented by `s`.
 - `s.toUpperCase()` is a new string whose characters are the uppercase equivalents of the character string represented by `s`.
 - `s.replace(t, u)` is a new string formed by first copying all of the characters of `s` and then replacing its first occurrence of substring `t` with substring `u`.
 - `s.replaceAll(t, u)` is a new string formed by first copying all of the characters of `s` and then replacing all occurrences of substring `t` with substring `u`.

InputStream

- Java uses `InputStream` to provide a system-independent view of an input data stream.
- `InputStream` methods provide access to the data as `byte` values.

File

- Java uses `File` to provide a system-independent view of a file.
- A `File` object has no methods to access file contents.
- Sample code for defining a `Scanner` variable based upon a user-supplied filename would be:

```
String s = stdin.next();
File f = new File( s );
Scanner fileIn = new Scanner( file );
```

System.in

- `InputStream` variable `System.in` represents the standard input stream; i.e.; where input should come from by default.
- `System.in` methods are not programmer friendly – they only provide access to the input in `byte` representation.

Scanner

- Java uses `Scanner` to provide a text view of an input data stream.
- Suppose `stream` is a `Scanner`.
 - `stream.next()` returns the next input value as a `String`.
 - `stream.nextInt()` returns the next input value as an `int`.
 - `stream.nextDouble()` returns the next input value as a `double`.
 - `stream.nextBoolean()` returns the next input value as a `boolean`.
 - `stream.nextLine()` returns the remainder of the current input line as a `String`.
 - `stream.hasNext()` returns a `boolean` value indicating whether there is any nonblank input left in the input stream.
 - `stream.hasNextLine()` returns a `boolean` value indicating whether there is another line of input (i.e., is there an unprocessed new line character).
 - `stream.hasNextInt()` returns a `boolean` value indicating whether the next input is integer. If there is no more input, the method returns false.
 - `stream.hasNextDouble()` returns a `boolean` value indicating whether the next input is decimal. If there is no more input, the method returns false.
 - `stream.hasNextBoolean()` returns a `boolean` value indicating whether the next input is `boolean`. If there is no more input, the method returns false.
- The following definition


```
Scanner stream = new Scanner( System.in );
```

makes `stream` a variable giving a `Scanner` view of standard input.

- The following definition using `File` variable `filename`

```
Scanner stream = new Scanner( filename );
```

 makes `stream` a variable giving a `Scanner` view of the file represented by `filename`.
- The following definition using `InputStream` variable `dataSource`

```
Scanner stream = new Scanner( dataSource );
```

 makes `stream` a variable giving a `Scanner` view of the input stream represented by `dataSource`.
- The following definition using `String` variable `s`

```
Scanner stream = new Scanner( s );
```

 makes `stream` a variable giving a `Scanner` view of the character string represented by `s`.

URL and URLConnection

- Java uses `URL` to provide as system-independent view of a `URL` – uniform resource locator; i.e., a web resource.
- A `URL` object has a method `openConnection()` for establishing a `URLConnection` between the program and a `URL`.
- A `URLConnection` object has a method `getInputStream()` for establishing a `URLConnection` between the program and a `URL`.
- A `URL` object has a convenience method `openStream()` for creating an `InputStream` view of a `URL`.
- Sample code for defining a `Scanner` variable based upon a user-supplied web site name would be:


```
String s = stdin.next();
URL u = new URL( s );
Scanner webIn = new Scanner( u.openStream() );
```

If statement

- The `if-else` statement uses a `boolean` expression to determine the next action executed by a program.
- An `if` statement is not a loop. Its action actions can only be executed once,
- An `if-else` statement has form


```
if ( Expression )
    Action1
else
    Action2
```
- where
 - The `else` part of the statement is optional.
 - The `if` action or `else` action is either a single statement or a block of statements within braces.

- It is highly recommended that an `if` action or `else` action always be a block of statements within braces
- The statements in an `if` action or `else` action can be any Java statement including `if-else` statements.
- If the test expression is true, the `if` action executes; otherwise, the `else` action executes.
- After the `if` action or `else` action completes, execution continues with the statement following the `if-else` statement.

Equality

- The operators `==` and `!=` test respectively whether two values are the same or different; e.g., two numbers or two object references.
- The object method `equals()` tests whether two objects are identical or different.

Doing assignments

- Normally each part of an assignment corresponds to something recently covered in class. If this does not seem to be the case, look again.

While statement

- The `while` statement uses a `boolean` expression to determine the next action executed by a program.
- A `while` statement has form


```
while ( Expression )
    Action
```
- where
 - The action is either a single statement or a block of statements within braces.
 - It is highly recommended that the action always be a block of statements within braces
 - The statements in the action can be any Java statement including `while` statements.
 - If the test expression is false, the action of the `while` statement is ignored and execution continues with the statement following the loop.
 - If the test expression is true, the action executes. After the action completes, the process of testing and action executing repeats until the test expression is false.

For statement

- The `for` statement uses a `boolean` expression to determine the next action executed by a program.
- A `for` statement has form


```
for ( OneTimeAction; Expression; AfterAction )
    Action
```
- where
 - The one time action is either a variable definition or assignment. The one time action is executed only once – right before the first evaluation of the test expression.
 - The scope of a variable defined in the one time action is limited to the `for` loop.
 - The action is either a single statement or a block of statements within braces.
 - It is highly recommended that the action always be a block of statements within braces
 - The statements in the action can be any Java statement including `for` statements.
 - If the test expression is false, the action and the after action are both ignored and execution continues with the statement following the loop.
 - If the test expression is true, the action executes. After the action completes, the after action executes. The process of testing and action and after action executing repeats until the test expression is false.

Loop processing

- Because the `while` and `for` only execute their action if the test expression is true; if the test expression is initially false, the action never executes.
- A loop that does not terminate is called an infinite loop
- `Scanner` has next message methods are useful when processing input. Suppose `stream` is a `Scanner`.
 - `stream.hasNext()` returns a `boolean` value indicating whether there is any nonblank input left in the input stream.
 - `stream.hasNextLine()` returns a `boolean` value indicating whether there is another line of input (i.e., is there an unprocessed new line character).
 - `stream.hasNextInt()` returns a `boolean` value indicating whether the next input is integer. If there is no more input, the method returns false.

- `stream.hasNextDouble()` returns a `boolean` value indicating whether the next input is decimal. If there is no more input, the method returns false.
- `stream.hasNextBoolean()` returns a `boolean` value indicating whether the next input is `boolean`. If there is no more input, the method returns false

Blocks and scope

- A block is a list of statements nested within braces.
- A method body is a block.
- A block can be placed anywhere a statement would be legal
- A block within another block is a nested block.
- A local variable is a variable defined within braces
- An identifier name cannot be reused within its block; i.e., a variable cannot be redefined within its block or nested blocks.
- A local variable can be used only in a statement or nested block occurring after its definition

Methods

- A method is named piece of code that can take parameters and produce a value.
- A request for a method to perform its action is called an invocation.
- When a method is invoked a new activation record is created. The activation records stores the values of its variables and parameters.
- Java has two types of methods – message methods and service methods.
- A message method sends a message to an object to perform one of its behaviors (actions) (e.g., `boolean` method `boolean` and `Scanner` method `boolean`). Without their objects, message methods do not make sense.
- Service methods don't require objects to carry out their tasks. A service method is useful on its own – it performs an action independent of any object (e.g., `Math.cos()`).
- Service methods precede their definitions with the keyword `static`; message methods do not.

Return

- Methods must specify a return type. The return type indicates what kind of value the method supplies back to the invocation. A method that does not supply a value must have a `void` return type.
- In a method definition, the return type immediately precedes the name of the method.
- A method that supplies a value uses a `return` statement to give the value.
- A `return` statement has one of two forms
`return;` or
`return expression;`
- A non `void` method must end its execution with a `return` statement of the second listed form. The expression following the keyword `return` gives the values to be supply back to the invocation.
- A `void` method need have a `return` statement – a `return` will automatically occur after the last statement in its body completes.
- A `return` statement is how one piece of code transfers values back to another piece of code.
- A `print()` or `println()` statement produces output for its user. The output is not communicated to other code.
- A `return` statement does not produce output and a `print()` or `println()` statement is not a `return` statement.

Formal parameters

- Java allows methods to have a list of parameters. The definition of the method must specify what parameters it requires. The specification is called the formal parameters.
- In a method definition, the formal parameters are enclosed within parentheses following the method name. If there are no formal parameters, the empty parentheses must still be there.
- The action of method is enclosed with braces following the formal parameter list.
- Formal parameters are considered local variables that are initialized with values of the actual parameters.

Actual parameters

- When a Java method with a nonempty formal parameter list is invoked there must be a list of values for initializing the formal parameters – the actual parameters

Formal and actual parameter interaction – value parameter passing

- At the start of a method invocation, the values of the actual parameters are copied into the activation record memory being kept for the formal parameters.
- The Java style of parameter passing is called *value parameter passing*. It gets this name because the values of the actual parameters are copied into the memory of the formal parameters. There is no other linkage between the values.
- Within a method, there is no access to the actual parameters from its invocation. Outside the method there is no access to its formal parameters.
- Because Java uses value parameter passing there is no way for a method invocation to change the value of an actual parameter; that is, because only a COPY of the actual parameter's value is passed into the method, if a method changes that value, only the COPY is affected. Any change is not visible outside the method. Once the method returns, the copy is forgotten.

Object parameters

- If a formal parameter is an object variable, then the formal parameter can change the attributes of the object it references.
- When the method completes, the actual parameter still references the same object it did before the invocation. However, any changes to that object made during the invocation remain.

Signature

- The signature of a method is name along with the types specified in the list of the formal parameters.

Overloading

- Java supports method overloading – different methods can have the same name.
- Method overloading is useful when methods are needed for similar tasks but with different types of parameter lists.
- Method name can be overloaded as long as its signature is different from the other methods of its class (i.e., different types or number of parameters).

Common looping ideas

- When processing all the values from a scanner stream
 - The test expression should use hasNext() if the values are strings, hasNextInt() if they are integers, and hasNextDouble() if they are decimals. Example, if stdin is your Scanner variable, then the while loop test expression should be stdin.hasNext()
 - while (stdin.hasNext()) {
 - ◆ String value = stdin.next();
 - ◆ Process value
 - }
- When summing n values.
 - Initialize your total variable to 0.
 - Initialize your counter variable to 0
 - While the counter variable is less than n
 - Add the next value to the total. (The particular value will depend on the list you have been told to sum.)
 - Increment the counter variable by 1.


```
int total = 0;
int counter = 0;
while ( counter < n ) {
    Get/make the value
    total = total + value;
}
```
- When multiplying n values.
 - Initialize your total variable to 1.
 - Initialize your counter variable to 0
 - While the counter variable is less than n
 - Multiply the next value to the total. (The particular value will depend on the list you have been told to multiply.)
 - Increment the counter variable by 1.


```
int product = 1;
int counter = 0;
while ( counter < n ) {
    Get/make the value
    total = total * value;
}
```
- When concatenating n strings.
 - Initialize your result variable to "".
 - Initialize your counter variable to 0
 - While the counter variable is less than n
 - Add the next value to the result. (The particular value will depend on the list you have been told to process .)

- Increment the counter variable by 1.


```
String result = "";
int counter = 0;
while ( counter < n ) {
    Get/make the value
    total = total + value;
}
```

ArrayList<T>

- Java uses `ArrayList<T>` to represent a dynamic list whose elements values of type `T`. For example, class `ArrayList<String>` can represent list whose elements are `String` values.
- The `ArrayList<T>` default constructor `ArrayList<T>()` configures a new `ArrayList<T>` to be an empty list. For example, the following definition makes `list` represent an empty list of type `ArrayList<T>`.


```
ArrayList<T> list = new ArrayList<T>();
```
- Suppose `list` is an `ArrayList<T>`. Also, suppose `v` is a value of type `T` and `i` is an `int`.
 - `list.add(v)` adds an element with value `v` to the end of `list`.
 - `list.clear()` removes all elements from `list`.
 - `list.contains(v)` returns whether value `v` equals one of the element values in `list`.
 - `list.get(i)` returns the value of the element with index `i` in `list`.
 - `list.indexOf(v)` returns the index of the first element in `list` with value `v`. If there is no such element, returns `-1`.
 - `list.isEmpty()` returns whether `list` has any elements.
 - `list.lastIndexOf(v)` returns the index of the last element in `list` with value `v`. If there is no such element, returns `-1`.
 - `list.remove(i)` removes the element with index `i` from `list`.
 - `list.remove(v)` removes the first element equal to `v` from `list`.
 - `list.set(i,v)` sets the element in `list` with index `i` to have value `v`.
 - `list.size()` returns the number of elements in `list`.
 - `list.toArray()` returns an `Object[]` array containing all the elements in `list`.

HashMap<S,T>

- Java uses `HashMap<S,T>` to represent a collection of associations. For example, class `ArrayList<String,Integer>` can associations

that map `String` values to integers. The associations are viewed as mappings from keys to values.

- The `HashMap<S,T>` default constructor `HashMap<S,T>()` configures a new `HashMap<S,T>` to have no associations. For example, the following definition makes `mapping` represent an empty list of associations from keys of type `S` to values of type `T`.


```
HashMap<S,T> mapping = new HashMap<S,T>();
```
- Suppose `mapping` is a `HashMap<S,T>`. Also, suppose `s` is a value of type `S` and `t` is a value of type `T`.
 - `mapping.clear()` removes all associations from `mapping`.
 - `mapping.containsKey(s)` returns whether there is association with key `s` in `mapping`.
 - `mapping.containsKey(s)` returns whether there is a key value associated with `t` in `mapping`.
 - `mapping.get(s)` returns what value key `s` is associated with in `mapping`.
 - `mapping.put(s,t)` adds the association of key `s` to value `t` to `mapping`.
 - `mapping.isEmpty()` returns whether `mapping` has any associations.
 - `mapping.remove(s)` removes the association for key `s` from `mapping`.
 - `mapping.size()` returns the number of associations in `mapping`.