List of common Java syntax errors

Capitalisation of key words

Since you get into the habit of writing class names in capital letters you will occasionally find yourself writing keywords such as `class` and`int` with a capital beginning letter. The compiler will object to this and will issue an error message which depends on which keyword was capitalised. The compiler will issue an error message such as:

```
Line nn: class or interface declaration expected
```

when, for example, you capitalise the keyword `class`.

Writing a string over a new line

Sometimes you will need to write a long string. A common error is to have a new line embedded in the string. The compiler will object to this and will issue an error message such as:

```
Line nn: ';' expected
```

When this happens the solution is to split the string into two, making sure that neither string has a new line in it, and concatenate them with +. Thus you might replace:

```
String s = "A very long string which just happens to go over the end of a
line and causes a problem with the compiler";
```

with:

```
String s = "A very long string which just happens to go over the end "+
"of a line and causes a problem with the compiler"
```

Missing brackets in a no-argument message

When you use a method which has no arguments you should place brackets after the name of the method. For example, if you have declared a method `carryOut` with no arguments and you want to send a message corresponding to the method to the object `objSend` then you should code this as:

```
objSend.carryOut()
```

rather than:

```
objSend.carryOut
```

The compiler will usually emit an error message of the form:

```
Line nn: Invalid expression statement
```

Forgetting to import a package

This one of the most common errors that inexperienced Java programmers make. If you forget to put the required `import` statement at the beginning of a program, then the compiler will respond with a message such as:

```
Line nn: Class xxxx not found in type declaration
```

Don't forget, though, that `java.lang` is imported automatically and, hence, does not need an import statement.

Treating a static method as if it were an instance method

Static methods are associated with messages sent to classes rather than objects. A common error is to send static method messages to objects. For example, in order to calculate the absolute value of an `int` value and place it into the `int` variable you should write:

```
int result = Math.abs(value);
```

rather than:

```
int result = value.abs();
```

This gives rise to a variety of syntax errors. The most common one is of the form:

```
Line nn: Method yyyy not found in class xxxx.
```

where *yyyy* is the name of the method and *xxxx* is the name of the class within which it is called.

Case-sensitive errors with classes

This is another category of error which is very common. Java is case sensitive so, for example, it will not recognise `string` as a valid type in the language as you should have written String. It will generate an error message of the form:

```
Line nn: Class xxxx not found in type declaration.
```

where xxxx is the name of the class which has not been given the correct capitalisation.

Case-sensitive errors with variables

It is also quite easy to miss the fact that variables are case sensitive. For example, you may have declared the variable `linkEdit` as an `int` and then tried to refer to `linkEdit` within a class. This gives rise to error messages of the form

```
Line nn: Undefined variable: xxxx
```

where xxxx is the name of the variable which has been mistyped.

Missing } brackets

This is a common programming error in *any* programming language and can be eradicated by means of a proper indentation scheme.

Missing class brackets

A common bracketing error that you will often make is to omit the final } bracket that delimits the end of a class.

Writing the wrong format for a class method

Class methods have the form:

```
ClassName.MethodName(Argument(s))
```

A common error is to forget the class name. If you do, then you will get an error message of the form:

```
Line nn: '}' expected
```

Specifying method arguments wrongly

When you define classes you should prefix each argument with the name of a scalar type or the name of an existing class. For example:

```
public void tryIt(int a, int b, URL c)
```

A common error that programmers from other languages make is to forget to prefix *every* argument with its type. For example, an erroneous version of the definition above would be:

```
public void tryIt(int a, b URL c)
```

This type of error will give rise to error messages of the form:

```
Line nn: Identifier expected
```

Forgetting the fact you should send messages to objects

This is a common error committed by programmers who have only recently changed to object-oriented programming. As an example of this consider the method `tryIt`, which has two `int` arguments and which delivers an `int` value. Assume that this method is involved in sending a message to an object `destination`. This should be written as:

```
int newVal = destination. tryIt(arg1, arg2)
```

where the arguments are ints which have been declared somewhere. A common mistake is to write this as:

```
int newVal = tryIt(destination, arg1,arg2)
```

This gives rise to error messages of the form:

```
Line nn: ')' expected
```

Assuming that == stands for value equality

== is used with scalars as a means of comparing values. However, when it is applied to objects then it compares addresses. For example, the `if` statement:

```
if(newObj1 == newObj2){
...
}
```

will execute the code denoted by the three dots *only* if the first object occupies the same address as the second object. If the objects occupied different addresses, but still had the same values for their instance variables, then it would evaluate to false. Unfortunately this does not give rise to any syntax errors, but will show up when any program containing the error is executed.

Omitting void in methods

When a method returns no result, but just carries out some action, you need to use the keyword void in front of the name of the method. If you do not use this keyword, then it will give rise to error messages of the form:

```
Line nn: Invalid method declaration; return type required
```

Omitting `break` from case statements

This is an error which is committed in both object-oriented and procedural languages. If you want the branch of a `case` statement to just finish and exit to the end of the `case` statement, then don't forget to include the `break` statement as the last statement in the branch. If you do not do this, then execution will continue with the next branch underneath the one in which the `break` statement was omitted.

Omitting the return in a method

When a method returns a value, then the body of the method should include at least one return statement which returns the right type of value. Failing to do this will generate an error message of the form:

```
Line nn: Return required at end of xxxx
```

where xxxx is the method which does not contain the return.

Making an instance variable private and then referring to it by name in another class

When you tag an instance variable as private you are not allowed to access it by name outside its class. The only way that you can access such instance variables is through methods which are declared in the class in which the instance variables are defined. This gives rise to error messages of the form:

```
Line nn: Variable xx in class xxxx not accessible from class yyyy
```

where xx is the private variable, xxxx is the class in which it is defined and class yyyy is the class in which it is referred to.

Using a variable before it is given a value

Again this is a common error found in both object-oriented and procedural languages. In Java, scalars are intialised to zero or some default value so there will be no error indication and any problems that arise will be signaled by erroneous results or some side effect such as an array going over its bounds. Objects will be initalised to null and any attempt to reference an uninitialised object will be caught at run time.

Assuming the wrong type of value is generated by a message

This is a common error to make when using the Java packages. A typical example is using a method which delivers a string that contains digits and treating it like an integer. For example, the method `getInteger` within `java.lang.Integer` delivers an Integer and any attempt to use that value as, say, an `int` will give rise to an error message of the form:

```
Line nn: Incompatible type for declaration can't convert xxxx to yyyy
```

Confusing prefix operators with postfix operators

This is an error that comes with any C-like language. Postfix operators such as ++ and -- deliver the old value of the variable to which they are applied, while prefix operators deliver the new value. Thus, if x is 45 and the statement:

```
y = ++x
```

is executed, then y and x both become 46. If the statement

```
y = x++
```

is executed, then y becomes 45, while x becomes 46. These errors will not be signalled at compile time, but will emerge during run time.

Forgetting that arguments are passed by reference to methods if they are objects

When an object is used as an argument to a method, then its address is passed over and not a value. This means that you can assign values to such arguments. If you treat them as values this will not strictly be an error, but will not be making use of the full facilities of an object-oriented programming language.

Forgetting that scalars are passed by value to methods

You cannot treat an argument which is a scalar as if it can be assigned to. This will not be signalled as a syntax error. However, it will show up as a run-time error when you write code which assumes that the scalar has been given a value by a method.

Misusing `size` when applied to strings and arrays

size is an instance variable associated with arrays and a method when associated with strings. If you mix them up by, for example writing:

```
arrayVariable.size()
```

or

```
stringVariable.size
```

then the first would generate an error message of the form:

```
Line nn: Method size() not found in class java.lang.Object
```

and the second would generate an error message of the form:

```
Line nn: No variable size defined in java.lang.String
```

Using a constructor which does not exist

You may use a constructor which has not been defined. For example, you may have a class `X` which has a one `int` constructor, a two `int` constructor and a three`int` constructor and yet you may have used a four `int` constructor. This would be picked up at compile time and an error of the form:

```
Line nn: No constructor matching xxxx found in class yyyy
```

would be generated, where *xxxx* is the signature of the constructor that you have tried using and *yyyy* is the name of the class which it should have been defined in.

Calling a constructor in a constructor with the same name

For example, you may have defined a class `X` with a two `int` constructor and a one `int` constructor and inside the two `int` constructor there is a reference to `X(argument)`. This will be flagged as an error and will generate an error message of the form:

```
Line nn: Method xxxx not found in yyyy
```

where *xxxx* is the name of the constructor and its arguments and *yyyy* is the name of the class which it is defined in. The solution is to use the `this` keyword.

Assuming that two-dimensional arrays are directly implemented in Java

This gives rise to erroneous code such as:

```
int [,] arrayVariable = new [10,20] int
```

This is illegal and will give rise to an errors of the form:

```
Line nn: Missing term
```

and:

```
Line nn: ']' expected
```

You can implement many-dimensional arrays in Java, but they are treated like single-dimension arrays which contain single-dimensional arrays which contain single dimension arrays, etc.

Treating a scalar like an object

Scalars such as `int` and `float` are not objects. However, sometimes you want to treat them as such, for example when you want to deposit them in a `Vector`, as in the code:

```
Vector vec = new Vector();
vec.addElement(12);
```

If you write code such as that shown above then it will give rise to syntax errors of the form:

```
Line nn: No method matching xxxx found in yyyy
```

where xxxx is the name of the method which is used and yyyy is the name of the class which expects an `Object`. The solution is to use the object wrapper classes found in `java.lang` to convert them to objects.

Confusing scalars and their corresponding object types

When you have scalars such as `int` it is easy to write code which assumes that they can be treated as if they were objects. For example, the code:

```
int y = 22;
Integer x = y;
```

will give rise to an error message of the form:

```
Line nn: Incompatible type for declaration. Can't convert xxxx to yyyy
```

where *xxxx* and *yyyy* are the classes involved.

Mistyping the header for the main method

When you want to execute a Java application you need to declare a method which starts with:

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

If you mistype any part of this line or miss out a keyword, then a run-time error will be generated. For example, if you miss out the keyword `static` then an error message of the form:

```
Exception in thread main.....
```

will be generated at run time.