# 9 APPENDIX

A ‘BAD’ STYLE PATTERNS ........ 57
I Over-complicating Boolean expressions ........ 57
II Usage of ‘magic’ Strings, numbers or characters ........ 57
III Overlong methods ........ 58
IV Long parameter lists ........ 58
V Creating infinite loops ........ 58
VI Looping unnecessarily ........ 59
VII Omitting the `default` case in a switch ........ 60
VIII Forgetting to `break` cases in a switch ........ 61
IX Returning temporary variables ........ 62
X Forgetting to initialise arrays ........ 63
XI Forgetting to initialise members of an Object array ........ 63
XII Referencing invalid Array indexes ........ 63
XIII Forgetting that Strings are immutable ........ 64
XIV Shadowing constants ........ 64
B IMPLEMENTED STYLE CHECKS ........ 65
I Over-complicating Boolean expressions ........ 65
II Usage of ‘magic’ Strings, numbers or characters ........ 65
III Overlong methods ........ 65
IV Long parameter lists ........ 66
VII Omitting the `default` case in a switch ........ 66
VIII Forgetting to `break` cases in a switch ........ 66
XIV Shadowing constants ........ 66
C DEVELOPER’S GUIDE ........ 67
I Working on KenyaEclipse ........ 67
II Caveats ........ 68
III The Style Guidance Module ........ 68
IV Writing StyleChecks ........ 68
D USER MANUAL ........ 75
I Prerequisites and Installation ........ 75
II The Kenya Perspective ........ 75
III Getting started ........ 76
IV The Kenya/Java editor ........ 78
V Running and Debugging your code ........ 80
VI Bad Style – the Style Guidance Module ........ 86
VII Advanced Editor features ........ 89
VIII Known Issues - Troubleshooting ........ 91
E TIME PLANS ........ 92
F LEARNING TO PROGRAM IN ECLIPSE ........ 95

# A ‘BAD’ STYLE PATTERNS

This is a listing of coding patterns (in no particular order) that can be considered ‘bad’ style in the sense that they may: (a-b more or less in order of importance)

a. cause the program to behave different from the expectation
b. hinder readability or comprehension of the code
c. violate principles of object oriented programming
d. adversely affect runtime performance in an obvious manner

## I Over-complicating Boolean expressions

*Category:* b

*Description:* For example, a student might write:

```
if( cond == true )
```

or

```
if( cond ) {
  return true;
} else {
  return false;
}
```

*Solution:* The above should be written as:

```
if( cond )
```

and

```
return cond;
```

*Rationale:* To improve conciseness and readability of the code

*Automation:* Possible

## II Usage of ‘magic’ Strings, numbers or characters

*Category:* c

*Description:* ‘Magic’ Strings are String literals that are hard coded in multiple places in the code. (similar for number and characters)

*Solution:* Declare the String/number/character as a constant

*Rationale:* To encourage code reuse and remove duplication

*Automation:* Possible

## III Overlong methods

| | |
|---|---|
| *Category:* | b, c (metric based) |
| *Description:* | Very long methods are hard to read. It may also indicate that the method is doing 'too much', i.e. things that lie outside its responsibility. |
| *Solution:* | Split method into smaller functional components. |
| *Rationale:* | To encourage code reuse and method specialisation (cohesion)<br>To improve conciseness and readability of the code |
| *Automation:* | not possible without user interaction |

## IV Long parameter lists

| | |
|---|---|
| *Category:* | b, c (metric based) |
| *Description:* | A method that takes many parameters may be inadequately placed and/or do 'too much'. |
| *Solution:* | Split method into smaller functional components or consider the use of objects that group co-occurring variables (e.g. use a `class Point` instead of using `int x` and `int y` together). |
| *Rationale:* | To encourage code reuse and method specialisation (cohesion)<br>To encourage use of Objects (-> Abstract Data Types)<br>To improve conciseness and readability of the code |
| *Automation:* | not possible without user interaction |

## V Creating infinite loops

*Category:* a, d

*Description:* Example of an infinite `while` loop:

```
int i = 5;
while(i==5) {
  //there is no break statement
  //there is no statement that alters i
}
```

Examples of an infinite `for` loop:

```
for(int i=0; i>=0; i++) {
  //there is no break statement
  //there is no break statement
}
```

*Solution:* Each of the above needs a revision of the loop condition or at least one case where a `break`, i.e. an escape from the loop is possible.

*Rationale:* To prevent erroneous program behaviour

*Automation:* Not easily achieved because of semantical issues

## VI Looping unnecessarily

*Category:* b, d

*Description:* A lot of lab assignments involve looping over an array once this data type has been introduced. Often this involves searching for an item in the array and returning its value, or whether it exists or not.

In this case students may forget that it is more efficient to return from the loop once the item has been found or rather, once the purpose of the loop has fulfilled its purpose.

For example:

```
boolean doesExist(int num, int[] index) {
  boolean result = false;
  for(int i=0; i<index.length; i++) {
    result |= (index[i] == num);
  }
  return result;
}
```

*Solution:* It is more efficient to return immediately, once it has been established that the item does exist:

```
boolean doesExist(int num, int[] index) {
  for(int i=0; i<index.length; i++) {
    if(index[i] == num ) {
      return true;
    }
  }
  return false;
}
```

*Rationale:* To encourage good programming practice

*Automation:* With restrictions as the semantics are not clear

## VII Omitting the `default` case in a switch

*Category:* a

*Description:* The mistake is to omit the default case from a switch statement and later assume that one of the cases was a match.

For example (with appropriate class and constant declarations):

```
Box bigOrSmall(int type) {
  Box b;
  switch(type) {
    case BIG: {
      b.size = BIGSIZE;
      break;
    }
    case SMALL: {
      b.size = SMALLSIZE;
      break;
    }
  }
  println("b's width is " + o.size.width);
  […]
  return b;
}
```

The invocation of `o.size.width` causes a `NullPointerException` to be thrown in the case where `type` was neither BIG nor SMALL.

*Solution:* The example above might return `null` immediately if ctype did not match BIG or SMALL:

```
Box bigOrSmall(int type) {
  Box b;
  switch(type) {
    case BIG: {
      b.size = BIGSIZE;
      break;
    }
    case SMALL: {
      b.size = SMALLSIZE;
      break;
    }
    default: {
      return null;
    }
  }
  println("b's width is " + o.size.width);
  […]
  return b;
}
```

*Rationale:* To encourage good programming practice
To prevent erroneous behaviour

*Automation:* Possible, but the user has to fill the `default` clause.

## VIII Forgetting to `break` cases in a switch

*Category:* a

*Description:* Forgetting the break clause in a switch case is often unintentional and happens because the programmer is in a hurry or in the beginning because of unfamiliarity with the language.

```
String bigOrSmall(int ctype) {
  String s;
  switch(ctype) {
    case BIG: {
      s = "  big";
    }
    case SMALL: {
      s = " small  ";
    }
    default: {
      s = "neither big nor small";
    }
  }
  println("s is " + s);
  […]
  return s;
}
```

The implementation above always returns “neither big nor small”. This kind of error may be hard to trace for a novice.

*Solution:* The above becomes:

```
String bigOrSmall(int ctype) {
  String s;
  switch(ctype) {
    case BIG: {
      s = "big";
      break;
    }
    case SMALL: {
      s = "small";
      break;
    }
    default: {
      s = "neither big nor small";
    }
  }
  println("s is " + s);
  […]
  return s;
}
```

Breaking in the default clause is not required, but might also be encouraged in order to maintain a level of consistency.

In cases where the fall through of a case is intentional, this should be explicitly marked by an appropriate comment:

```
String bigOrSmall(int ctype) {
  String s;
  switch(ctype) {
    case BIG: {
      //fall through
    }
    case SMALL: {
      o = "either big or small";
      break;
    }
    default: {
      s = "neither big nor small";
    }
  }
  println("s is " + s);
  […]
  return o;
}
```

*Rationale:* To prevent erroneous behaviour
To encourage comment usage to explain ambiguous code

*Automation:* User interaction is required, but can in general be achieved

## IX Returning temporary variables

*Category:* b

*Description:* This applies only to cases where a temporary variable is created, assigned a value in a block statement (such as switch) and is finally returned without using it for any other purpose than this return:

```
boolean cond;
[…]
int temp
if(cond) {
  temp = 50;
} else {
  temp = 1000;
}
return temp;
```

*Solution:* If the result of the method is known, return immediately:

```
boolean cond;
[…]
if(cond) {
  return 50;
} else {
  return 1000;
}
```

*Rationale:* To improve code readability

*Automation:* Possible

## X Forgetting to initialise arrays

*Category:* a

*Description:* Forgetting to initialise arrays and then referencing their members results in an exception to be thrown at runtime. The error can be hard to trace by a novice.

*Solution:* Check whether arrays are initialised before their first use.

*Rationale:* To prevent erroneous program behaviour

*Automation:* Possible

## XI Forgetting to initialise members of an Object array

*Category:* a

*Description:* It frequently happens that an array is initialised, but the members are not, resulting in a NullPointerException to be thrown:

```
Point[] points = new Point[3];
[…]
points[0].x = 5;
```

*Solution:* Check whether the array contents are initialised.

*Rationale:* To prevent erroneous program behaviour

*Automation:* Potentially complex

## XII Referencing invalid Array indexes

*Category:* a

*Description:* Valid array indexes are any integer from 0 and array.length-1 inclusive. Students may make the mistake of forgetting that Java array indexing starts at 0 and construct a loop that iterates over the end of the array:

```
int[] nums = new int[5];
[…]
for(int i=1; i<=nums.length; i++) {
  nums[i]+=2;
}
```

This code throws an ArrayIndexOutOfBoundsException on the last loop iteration. Furthermore, the index 0 is not included in the operation.

*Solution:* The indexes should be adjusted to range from 0 to the length of the array (excluding):

```
int[] nums = new int[5];
[…]
for(int i=0; i<nums.length; i++) {
  nums[i]+=2;
}
```

*Rationale:* To prevent erroneous behaviour

*Automation:* Possible

## XIII Forgetting that Strings are immutable

*Category:* a

*Description:*

```
const String checked = "checked: ";

void main(String[] args) {
  String f = args[0];
  prependChecked(f);
  […]
}

void prependChecked(String s) {
  s = checked + s;
}
```

Here the user expects the argument String f to be changed to its own value prefixed with “checked: ”. However, since Strings are immutable, a method call to prependChecked() actually has absolutely no effect.

*Solution:* A solution would be to change the return type of prependChecked() to String and to return the modified String. The return value should be assigned to the argument:

```
const String checked = "checked: ";

void main(String[] args) {
  String f = args[0];
  f = prependChecked(f);
  […]
}

String prependChecked(String s) {
  s = checked + s;
  return s;
}
```

*Rationale:* To prevent erroneous program behaviour

*Automation:* Possible

## XIV Shadowing constants

*Category:* a, b

*Description:* This occurs if the user declares a local variable with the same name as a constant. The local declaration has precedence and thus prevents usage of the constant, while the user might think the opposite.

*Solution:* The easiest way to avoid this situation is to rename the local variable and all its occurrences to remove the clash in naming.

*Rationale:* To prevent unanticipated behaviour
To avoid the ‘need’ for unreadable code

*Automation:* Possible

# B IMPLEMENTED STYLE CHECKS

This is a listing of all implemented StyleCheckers by proposals as defined in Appendix A. All classes and IDs are prefixed with `kenya.eclipse.style.checks`. The default value for the 'defaultEnabled' flag is `true` and thus omitted in most cases.

## I Over-complicating Boolean expressions

| | |
|---|---|
| **name** | Check for overcomplicated booleans |
| **ID** | `bool.BooleanExpressionChecker` |
| **class** | `bool.BooleanExpressionChecker` |

| | |
|---|---|
| **name** | Check for notorious 'If return true else return false' |
| **ID** | `bool.IfReturnElseReturnChecker` |
| **class** | `bool.IfReturnElseReturnChecker` |

| | |
|---|---|
| **name** | Check for empty 'if' blocks |
| **ID** | `bool.EmptyIfElseChecker` |
| **class** | `bool.EmptyIfElseChecker` |

## II Usage of 'magic' Strings, numbers or characters

| | |
|---|---|
| **name** | Magic string Checker |
| **ID** | `magic.MagicStringChecker` |
| **class** | `magic.MagicStringChecker` |

| | |
|---|---|
| **name** | Magic char Checker |
| **ID** | `magic.MagicCharChecker` |
| **class** | `magic.MagicCharChecker` |

| | |
|---|---|
| **name** | Magic int/double Checker |
| **ID** | `magic.MagicNumberChecker` |
| **class** | `magic.MagicNumberChecker` |

## III Overlong methods

| | |
|---|---|
| **name** | Check for method length > 10 |
| **ID** | `metrics.MethodLengthChecker10` |
| **class** | `metrics.MethodLengthChecker` |
| **defaultEnabled** | `false` |
| **custom attributes** | length = 10 |

| | |
|---|---|
| **name** | Check for method length > 20 |
| **ID** | `metrics.MethodLengthChecker20` |
| **class** | `metrics.MethodLengthChecker` |
| **defaultEnabled** | `true` |
| **custom attributes** | length = 20 |

| | |
|---|---|
| **name** | Check for method length > 50 |
| **ID** | `metrics.MethodLengthChecker50` |
| **class** | `metrics.MethodLengthChecker` |
| **defaultEnabled** | `false` |
| **custom attributes** | length = 50 |

## IV Long parameter lists

| | |
|---|---|
| **name** | Check for long parameter lists ( > 8 ) |
| **ID** | `metrics.ParameterLengthChecker` |
| **class** | `metrics.ParameterLengthChecker` |
| **defaultEnabled** | `true` |
| **custom attributes** | length = 8 |

## VII Omitting the `default` case in a switch

| | |
|---|---|
| **name** | Check for omission of default case in switch blocks |
| **ID** | `swit.DefaultOmissionChecker` |
| **class** | `swit.DefaultOmissionChecker` |

## VIII Forgetting to `break` cases in a switch

| | |
|---|---|
| **name** | Check for break statement omission in switch cases |
| **ID** | `swit.BreakOmissionChecker` |
| **class** | `swit.BreakOmissionChecker` |

## XIV Shadowing constants

| | |
|---|---|
| **name** | Check for shadowed constants |
| **ID** | `scope.ShadowedConstantsChecker` |
| **class** | `scope.ShadowedConstantsChecker` |

# C DEVELOPER'S GUIDE

This guide provides details of how to extend the *KenyaEclipse* system and in particular how to create new StyleChecks.

## I Working on KenyaEclipse

### Requirements

- Eclipse IDE version 3.0 or later with platform SDK installed (see caveats)
- KenyaEclipse project source code available from http://doc.ic.ac.uk/kenya
- KenyaEclipse project report, chapter 5.3 - Style Guidance Module (SGM), for a more detailed and functional description of implementation than this guide offers

### Package functionality listing

These are all packages of KenyaEclipse (excluding core Kenya packages) and the functionality they contain:

| | |
|---|---|
| kenya.eclipse. | root package, contains main plugin class and constants |
| ast. | AST handling, dynamic binding resolution, etc. |
| buildext. | linkage between compiler and postBuildAnalyser extension |
| debug. | launching: run and debug classes |
| bridge. | bridges between StackMachine and the debug model |
| launcher. | Kenya launcher (declared in plugin.xml) |
| model. | the debug model, i.e. threads, stackframes, breakpoints, … |
| sourcelookup. | how the Kenya source code is looked up |
| ui. | UI related, e.g. InsertEOFAction |
| multieditor. | everything pertaining to the editor |
| java. | Java part of the editor |
| kenya. | Kenya part of the editor (including actions) |
| completion. | code completion processor and related (ctrl+space) |
| correction. | code correction processor (e.g. for style) (ctrl+1) |
| occurrences. | some occurrence finder stuff, main occ. code is in KenyaEditor |
| refactoring. | these classes are similar to JFace's TextEdit, etc. classes |
| util. | utilities, e.g. LocationUtils, AutoIndentStrategy |
| text. | some code scanners/readers and formatter (unused) |
| util. | utilities, e.g. JavaInput and provider, KenyaHelperThread |
| natures. | contains all 'natures' (e.g. KenyaNature) |
| preferences. | contains preference related classes |
| style. | Style Guidance Module - base classes: registry, manager, etc. |
| checkerimpl. | base classes for kenyaStyleCheckers extension point |
| checks. | implementation of style checkers |
| properties. | e.g. StylePropertyPage for configuration |
| ui | e.g. content providers for use in StylePropertyPage |
| ui. | general UI classes |

## II Caveats

### Eclipse version discrepancies

This concerns 3.0 -> 3.1 compatibility issues.
The KenyaEclipse version for Eclipse 3.0 uses inofficial API from `debug.ui`. The class `org.eclipse.debug.internal.ui.stringsubstitution.StringVariableSelectionDialog` is directly referenced by `kenya.eclipse.debug.ui.launchconfig.ArgumentsTab`. The class has been promoted to official API in the 3.1 release and as thus its package has changed. If developing for Eclipse 3.1 and above you must make sure that the import reads `org.eclipse.debug.ui.StringVariableSelectionDialog`.

The Eclipse milestone build 3.1M5 has introduced an 'insert EOF' feature for the Console View. It is activated by a platform specific keyboard command (ctrl-d or ctrl-z). For Eclipse 3.1, the insertEOF action could thus be removed and the manual updated accordingly.

## III The Style Guidance Module

The module is exclusively defined in `kenya.eclipse.style`. The extension point `kenya.eclipse.postBuildAnalyserFactories` is used to connect the module to the editor. The multieditor class `kenya.eclipse.multieditor.EditingWindow` manages this extension point.

For a detailed description of the implementation you should either consult the source code for the classes above or obtain a copy of the project report, which contains more details about the extension point and its management.

StyleCheckers are governed by the `kenya.eclipse.styleCheckers` extension point, which is managed by `kenya.eclipse.style.StyleManager`.

## IV Writing StyleChecks

StyleChecks can be supplied in a separate plug-in and need not be packaged in the same archives as KenyaEclipse. The only requirement for this is the availability of the KenyaEclipse plug-in during development and at runtime.

A StyleChecker is the implementation of an extension point and thus has two components: firstly the class implementing the actual check, and secondly the declaration in the plugin.xml descriptor file.

### Declaring and implementing a StyleChecker

The extension point itself is declared as follows:

```
<extension-point
  id="styleCheckers"
  name="Kenya Style Checkers"
  schema="schema/styleCheckers.exsd"
/>
```

Extension point description:

***Identifier:** kenya.eclipse.KenyaStyle.styleCheckers*

***Configuration Markup:***

```
<!ELEMENT extension (KenyaStyleChecker)>
<!ATTLIST extension
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED
  point CDATA #REQUIRED>

<!ELEMENT KenyaStyleChecker (customAttribute)>
<!ATTLIST KenyaStyleChecker
  class          CDATA #REQUIRED
  id             CDATA #REQUIRED
  name           CDATA #REQUIRED
  defaultEnabled (true | false) >
```

- **class** - Fully qualified name of a class that implements IStyleChecker
- **id** - unique identifier for the checker
- **name** - A name which will be used in the preference panel, a short description would be good.
- **defaultEnabled** - whether or not this check should be enabled by default, the default value is 'true'.

```
<!ELEMENT customAttribute EMPTY>
<!ATTLIST customAttribute
  name  CDATA #REQUIRED
  value CDATA #REQUIRED>
```

- **name** - name of the attribute (map key)
- **value** - value of the attribute (map value)

Each extension encompasses one or more ‘KenyaStyleChecker’s, which require attributes as above.

The name is actually displayed on the configuration page and should be chosen to be a short description of the checker.
The defaultEnabled flag specifies whether the check is enabled by default. This may be useful if providing a StyleChecker with custom attributes (see below), where the class can be reused multiple times, but only one of them should be enabled initially.
It is usually sufficient to use the same value for id as for the class, but note that if the same class is used, then the id has to be unique for each declaration.
A good example if this is the provided MethodLengthChecker.

Finally, KenyaStyleCheckers may contain multiple ‘customAttribute’s, which are mappings of keys to values. These can be used to create reusable StyleCheckers.

Here, we will take the ‘Long parameter list’ checker as an example and see how it is declared and implemented in the plug-in. Although the extension point definition seen above looks complex, it boils down to only a few lines.

## Declaration

The checker is declared in the plugin.xml file as follows:

```
<KenyaStyleChecker
    class="kenya.eclipse.style.checks.metrics.ParameterLengthChecker"
    name="Check for long parameter lists ( &gt; 8 )"
    id="kenya.eclipse.style.checks.metrics.ParameterLengthChecker8">
    <customAttribute name="length" value="8" />
</KenyaStyleChecker>
```

**Note:**
**id** is the unique identifier. Since this StyleChecker can be configured with different values for its ‘length’ attribute, that value has been appended
**customAttribute** with name ‘length’ and the value ‘8’.

The **defaultEnabled** attribute is missing from the declaration since its default value is `true`, although it might be included for added clarity.

And that is it. This is all the markup required to integrate a StyleChecker with the framework. Only the implementation needs to be added.

## Implementation

The first thing to note is that, while the extension point only requires the implementation to inherit from `IStyleChecker`, there is an abstract implementation available, providing various useful methods for use by its subclasses.

We can create our class, extending `AbstractStyleChecker`, which requires us to implement the two methods:

```
void configure(java.util.Map)
void performCheck(mediator.ICheckedCode, IFile file).
```

`configure` is called first and passes all custom attributes declared using the customAttribute tag to the instance of the checker as a map, mapping ‘name’s to ‘value’s. In our implementation, we retrieve the value for the ‘length’ attribute and store it in a field:

```java
private int fMaxLength = Integer.MAX_VALUE; //default is 'infinity'

public void configure(Map customAttributes) {
  String ln = (String)customAttributes.get("length");
  if(ln!=null) {
    try {
      int length = Integer.parseInt(ln);
      fMaxLength = length;
      //retrieve the length from the configuration
    } catch(NumberFormatException e) {
      //ignore (if fails, we will be using the default value)
    }
  }
}
```

The implementation of `performCheck` can be staged in 3 phases. First, we will perform a search to find all method declarations, since this is the point where we will find parameter lists. Second, we can single out the methods that take more parameters than `fMaxLength` (see above) and can then in the third phase take action (resolutions and highlighting).

### Phase 1

Obtaining method or function references from the code object is simple:

```
IFunction[] funs = code.getFunctions();
```

That’s phase 1 done.

Phase 2

We can store the ‘offenders’ in a collection to be able to treat them later. First we need to find these by examining the methods we just found:

```
//we will need this later
IDocument doc = getDocument(file);

//lengths will contain the offending method's location
// and the number of arguments they accept
HashMap lengths = new HashMap();

//we first examine each function in turn
for(int i = 0; i < funs.length; i++) {
  IFunction function = funs[i];
  AFuncDecDeclaration decl = function.getDeclarationNode();
  //the following is cheating as it
  // requires implicit knowledge of the AST.
  // the 'proper' way is to use a
  // visitor that counts the number of parameters
  AFormalParamList paramlist = (AFormalParamList)
                              decl.getFormalParamList();
  if(paramlist==null) {
    continue; //no parameters
  }
  Node n = paramlist.getTypeName();
  List list = paramlist.getCommaTypeName();

  int length = (n==null)
    ?0                   //if n is null, length is 0
    :1 + (               //otherwise 1 for n, and add..
      (list==null)
          ?0             //..0 for a null list
          :list.size()  //..or its length
       );

  if(length>fMaxLength) { //test against max length
    //we actually wish to highlight the parameter list
    ISourceCodeLocation loc
      = AdvancedPositionFinder.getFullLocation(paramlist, doc);
    lengths.put(loc, new Integer(length));
  }
}
```

Note that we used a map of locations to number of parameters to store the ‘offenders’. To calculate the position of the parameters in the file, we used superclass functionality to obtain the corresponding `IDocument`.

Phase 3

The final step is to use the gathered information to create the annotations in the editor.

```
//can now go through 'offenders' and create the appropriate markers
//this must be done in an IWorkSpaceRunnable
final Set offences = lengths.entrySet();
IWorkspaceRunnable runnable = new IWorkspaceRunnable() {
  public void run(IProgressMonitor monitor) throws CoreException {
    for(Iterator it = offences.iterator(); it.hasNext();) {
      Map.Entry entry = (Map.Entry)it.next();
      //retrieve the data from the map
```

```
       ISourceCodeLocation loc
           = (ISourceCodeLocation)entry.getKey();
       Integer length = (Integer)entry.getValue();

       String msg
           = MessageFormat.format(MESSAGE, new Object[]{length});

       //super can do the marker creation
       createKenyaStyleMarker(file, loc, msg);

     }
   }
};
```

where MESSAGE is a constant declared as:

```
public static final String MESSAGE =
  "This method takes {0} parameters. " +
  "\nTry splitting the method into " +
  "smaller functional parts or creating" +
  "\na class to group closely related parameters.";
```

The {0} is replaced by the length of the parameter list as obtained from the map by the format instruction. The message is used as a label for the created marker and is the text shown, for instance, in the problem view for that particular style warning.

Finally, this runnable needs to be executed. Again, the abstract superclass provides a method for this and we simply call:

```
//execute the runnable
runMarkerUpdate(runnable);
```

We have seen:

- how easy it is to write a StyleChecker
- that for more complicated checkers you will need to use the AST and visitors
- that `AbstractStyleChecker` provides various helpful methods

We have not seen:

- How to create resolutions

## Style Resolutions

Resolutions can be created during the execution of the `IWorkspaceRunnable`. The idea is to instantiate and populate a map, which is defined in the field `fResolutionMap`. It can be instantiated at any time and populated as required. This map is returned to the framework by a call to the predefined method `Map getMarkerResolutionMap()`. The map is expected to be populated once the runnable returns and maps from `IMarker` to `IMarkerResolution[]`.

An `IMarker` is returned from `createKenyaStyleMarker`, while the Resolution can be created as an inner class or extension of `StyleWarningResolution`. This contains all information necessary to perform the resolution.

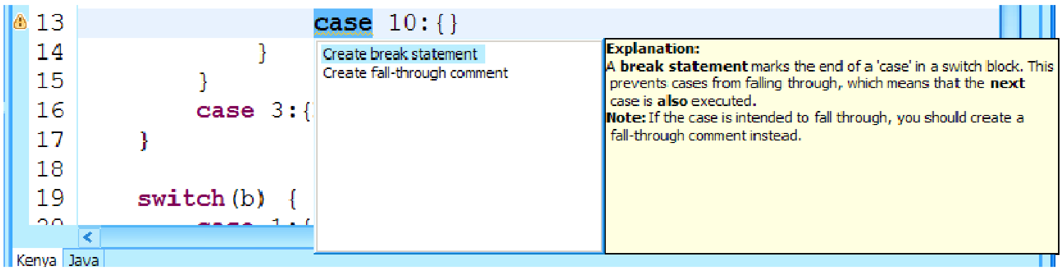
Use the field `fOperation` in `StyleWarningResolution`, which is an instance of `DocumentModificationOperation` (`kenya.eclipse.multieditor.kenya.refactoring`), to define text operations that will perform the resolution. If you do not want a modification at all, simply don't touch `fOperation`. It is still possible to show the resolution dialog.

Here is an example of a `StyleWarningResolution` implementation:

```java
class BreakOmissionMarkerResolution1 extends StyleWarningResolution {
  public BreakOmissionMarkerResolution(IMarker marker) {
    super(null, marker, new DocumentModificationOperation());
    fLabel = "Create break statement";
    initDescription();
    init(marker);
  }
  private void initDescription() {
    fDescription = "<p><b>Explanation:</b></p>" +
     "<p>A <b>break statement</b> marks the end of a 'case' in a " +
     "switch block. This prevents cases from falling through, which " +
     "means that the <b>next</b> case is <b>also</b> executed.</p>" +
     "<p><b>Note:</b> If the case is intended to fall through, you " +
     "should create a <em>fall-through comment</em> instead.</p>";
  }
  private void init(IMarker marker) {
    IDocument doc = getDocument((IFile)marker.getResource());
    if(doc==null) { return; }
    //find the place where we are supposed to insert our generation
    int offset = marker.getAttribute(FIX_OFFSET, -1);

    if(offset<0) { return; } //can't continue if no such place defined

    String prefix; //line indent
    String postfix; //next line indent

    […] //calculate prefix and postfix

    String insertion = prefix + "break;" + postfix;

    fOperation.addOperation(
         DocumentTextOperation.newTextInsertion(offset, insertion));
  }
}
```

Three different text operations are defined, namely insertion, deletion and replacement. You will also have noticed that the description is written using HTML markup. Although not all tags are supported, this provides a way of formatting the displayed text.

Another point to note is the mapping to `IMarkerResolution[]`. Using an array allows us to specify more than one solution to the problem. In the actual implementation of the `BreakOmissionChecker`, another resolution is implemented, allowing the addition of a fall-through comment instead of a `break` statement. All resolutions from the array are added to the proposal box as shown below. Notice also the formatting resulting from the usage of HTML.



```
                       case 10:{}
                  }
             }
             case 3:{
        }

        switch(b) {
```

Create break statement
Create fall-through comment

**Explanation:**
A **break statement** marks the end of a 'case' in a switch block. This prevents cases from falling through, which means that the **next** case is **also** executed.
**Note:** If the case is intended to fall through, you should create a fall-through comment instead.

# D USER MANUAL

If concepts occurring in this manual are not sufficiently explained, please use the Eclipse **Workbench User Guide** to complement.

## I Prerequisites and Installation

KenyaEclipse is fully compatible with Eclipse version 3.0 and above on all operating systems. To be able to use the plug-in, you need to install a version of Eclipse corresponding to your operating system.

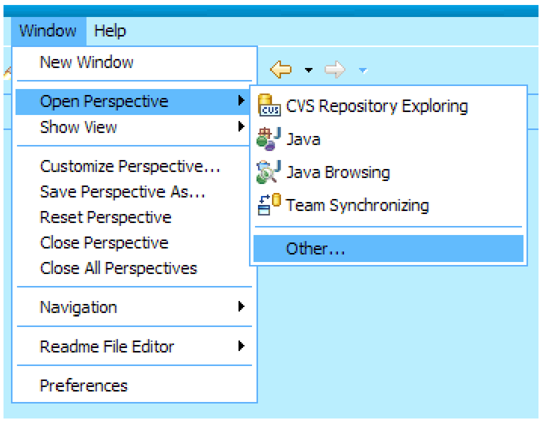
To install KenyaEclipse, obtain the version that is intended for your version of Eclipse, there is one version for Eclipse 3.0 and a separate version for Eclipse 3.1.
KenyaEclipse is distributed in a zip archive (a file with the ending '.zip') and you will require WinZip or an equivalent decompression utility to view the contents.

Using the utility, extract the archives contents to the directory in which you installed Eclipse. The next time you start Eclipse, KenyaEclipse's functionality will be available for your use.

In order to install this manual into the Eclipse help system, please proceed exactly as above with the archive containing the KenyaEclipseHelp plug-in. It will then be accessible via the Help -> Help Contents link in Eclipse.

## II The Kenya Perspective

In order to work with Kenya, you will need to switch to the Kenya Perspective. You can do this by clicking on the menu item **Window** and selecting **Open perspective**.
*Should* a link to the Kenya Perspective not appear in the submenu, then please select **Other** as shown below and choose Kenya from the list.



In consequent uses, the Kenya Perspective should now be selected, but can always be activated using the same steps described here.

The Kenya Perspective offers access to all functionality that relates to Kenya, such as the Kenya editor and run and debug support for Kenya programs. The following few chapters explain how to make use of this perspective in order to program with Kenya.
You will notice that you are using the Kenya Perspective by the changed window title.

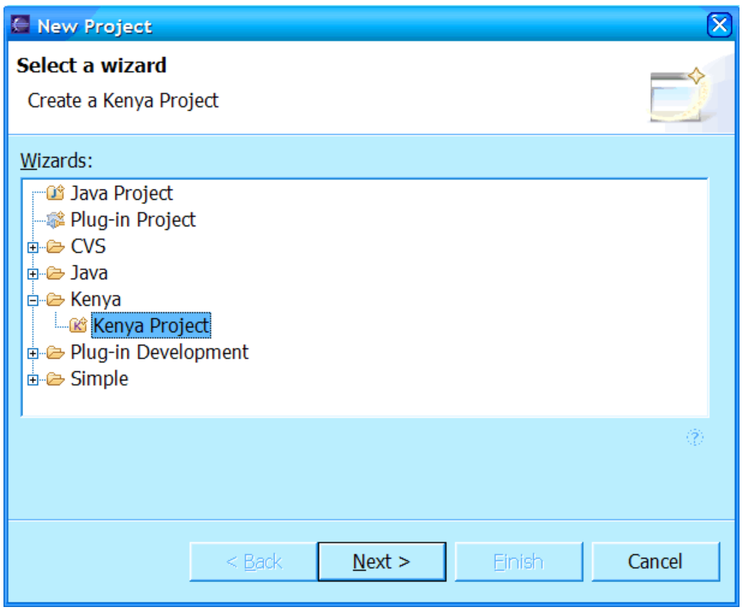
# III Getting started

This chapter describes how to get to the point where you can start writing your first piece of code.

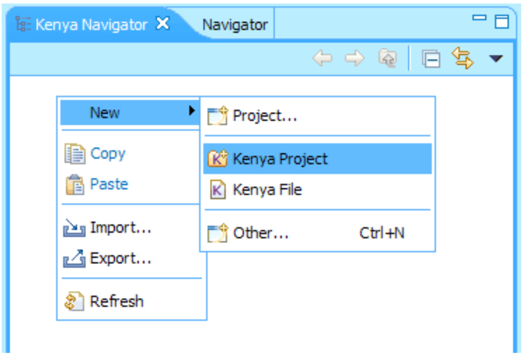
## Creating a Kenya Project

In Eclipse everything is done in Projects. There are different kinds of projects for different kind of purposes, such as Java Projects for Java or Kenya Projects for Kenya. All your work with Kenya should be put into Kenya Projects.

1. Inside Eclipse select the menu item **File > New > Project...**. to open the **New Project** wizard. Select **Kenya Project** in the **Kenya** folder then click **Next**.

   **Alternatively**, if the Kenya perspective was already open, then you can right click into the Kenya Explorer and select **New -> Kenya Project** from the menu.

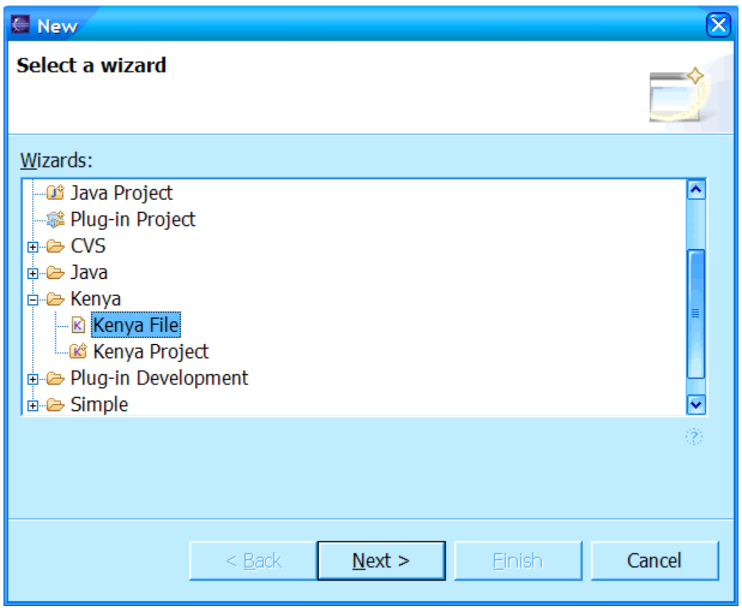
   

2. On the next page, type a name in the **Project name** field and click **Finish**. The Kenya perspective automatically opens inside the workbench with the new Kenya project in the Kenya Explorer.
   The project is stored on disc in your workspace folder. The Kenya Explorer presents this as a logical location in which you can create, delete and modify files.

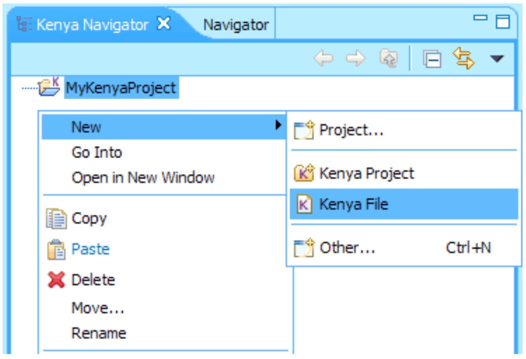
## Creating a new Kenya file

The Kenya file is the smallest unit inside KenyaEclipse. Each program you write resides in its own file. The **New Kenya File** wizard makes it easy to create a new program in Eclipse.

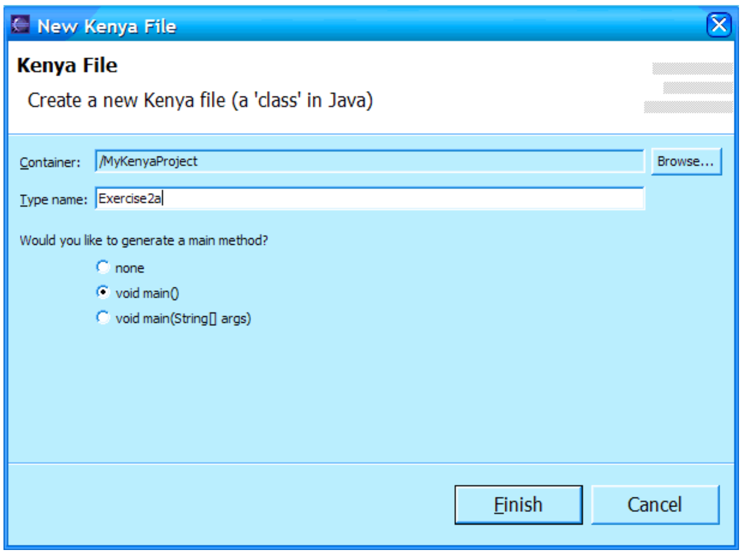
1. Inside Eclipse select the menu item **File > New > Other** to open the **New** wizard. Select **Kenya File** in the **Kenya** folder then click **Next**.



**Alternatively**, if the Kenya perspective was already open, then you can right click into the Kenya Explorer and select **New -> Kenya File** from the menu.



2. The next page asks you to select a container, which is by default the last selected folder or project on the Explorer or navigator.
You are also asked to enter a **Type** name. This is both the filename and what is in Java known as the Class name. You will notice that certain names are not permitted by convention and you will be given a reason why.
The last thing asked is whether you would like to generate a **main method**. Since all Kenya programs require a main method to be present, the default is to create one without arguments. The main method will be automatically inserted into the new file, so that you do not need to type it yourself. This is just one of the ways in which an IDE can speed up your development.

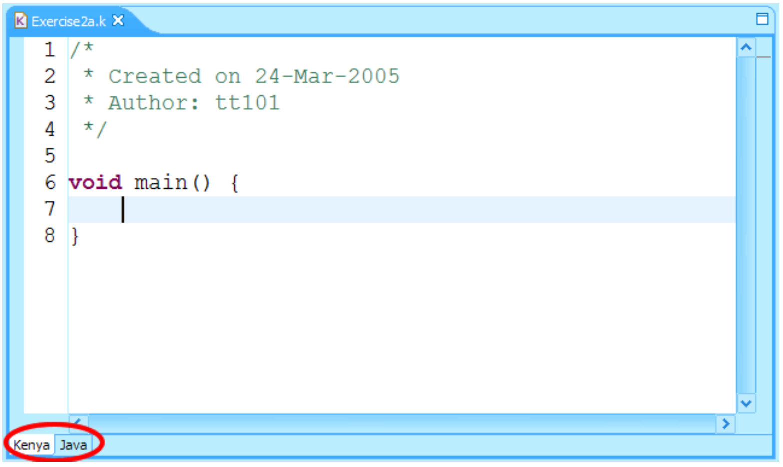

New Kenya File
Kenya File
Create a new Kenya file (a 'class' in Java)
Container: /MyKenyaProject
Browse...
Type name: Exercise2a
Would you like to generate a main method?
none
void main()
void main(String[] args)
Finish
Cancel

3. The new file will open in the Kenya/Java editor. If you have not yet switched to the Kenya Perspective, you should do it now.
   You are now ready to start programming your first Kenya application using KenyaEclipse. How to use the editor and its features is explained in the next chapter.

# IV The Kenya/Java editor

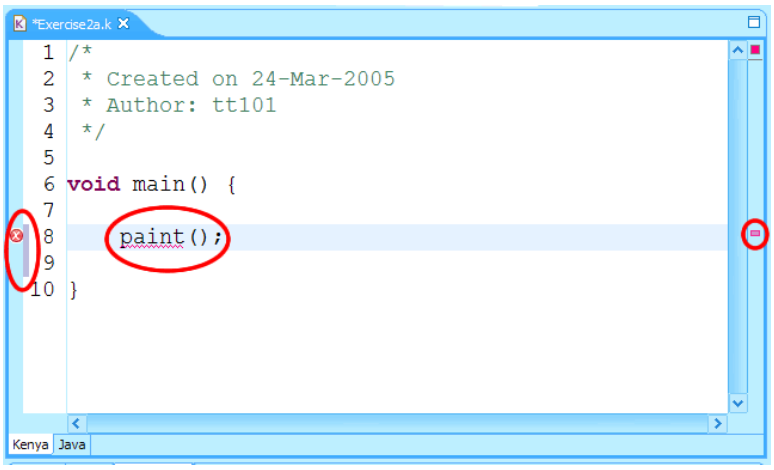
## Layout

The editor consists mainly of two parts. One is where you type Kenya code, the other is where you can view the corresponding Java code (providing your code is error-free). To switch between the views, all you have to do is use the tabs at the bottom of the editor.



```
/*
 * Created on 24-Mar-2005
 * Author: tt101
 */

void main() {
    
}
```

The editor highlights comments, keywords, Strings etc in different colours as you type. It will also highlight errors while you type. The following picture shows a compilation error. In this example the function ‘paint’ is not defined.



*Exercise2a.k

```
 1 /*
 2  * Created on 24-Mar-2005
 3  * Author: tt101
 4  */
 5
 6 void main() {
 7
 8     paint();
 9
10 }
```

Kenya Java

There are four areas of attention in this screenshot:

- red squiggles in the code (simply highlight the error)
- a red cross on the **left**. That area is called the **Annotation Ruler**
- a red marker on the **right**. That area is called the **Overview Ruler**
- a grey bar next to lines 8 and 9. This is called the **Change Ruler**

**Annotation Ruler**
The annotation ruler shows icons for different annotations. These could be errors, warnings, bookmarks, breakpoints, etc., some of which will be covered later. Clicking on the icon will normally highlight the portion of code that is relevant to the annotation.

**Overview Ruler**
The overview ruler shows small coloured blocks with different colours for different types of annotations, similar to the annotation ruler. The difference is that the overview ruler is scaled to the size of the document. This means that, while the annotation ruler's visibility is limited to the current part of the code you are looking at, the overview ruler always has the same scale, so you can use it to quickly navigate to an annotation that is not showing on the screen right now.

**Change Ruler**
The change ruler gives an indication of what has changed in the document since the last time it was saved. This can be quite powerful when trying to track changes that broke the functionality of the program.

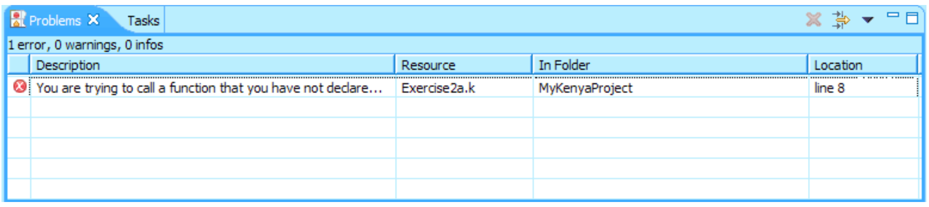
**Hovering** over each of these rulers will give you more information about why they exist. They might contain an error message or the contents of a bookmarked section of code.

**Right-clicking** allows you to access some settings, add bookmarks, tasks, etc. these actions are also available from the **Edit** menu item.

The three rulers will be of importance again, when we consider advanced editor features.

## Problem view

Problems also have a place in their own view, which resides at the bottom of the screen:



Problems Tasks

1 error, 0 warnings, 0 infos

| | Description | Resource | In Folder | Location |
|---|---|---|---|---|
| ⊗ | You are trying to call a function that you have not declare… | Exercise2a.k | MyKenyaProject | line 8 |

Errors are displayed here even if the file is closed and can be used for quick navigation.

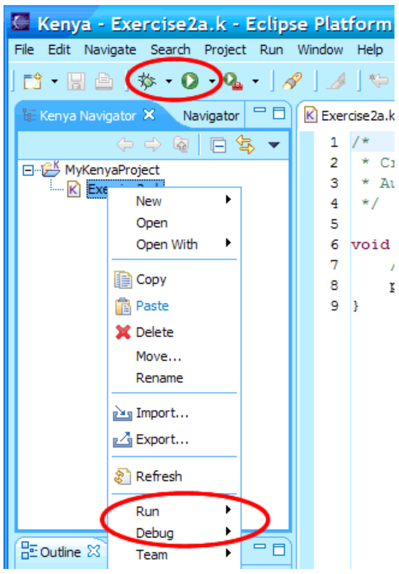
# V Running and Debugging your code

Look at this piece of code:

```
/*
 * Created on 24-Mar-2005
 * Author: tt101
 */

void main() {
      //print the famous words...
      println("Hello World!");
}
```

Its purpose should be quite clear, but of course this isn't worth anything if it is not possible to execute it. So how do we go about that? The first 3 steps apply to both running and debugging.
**Make sure** you also read the chapter about the **Console View**, it is quite important for programming using I/O (Input/Output).

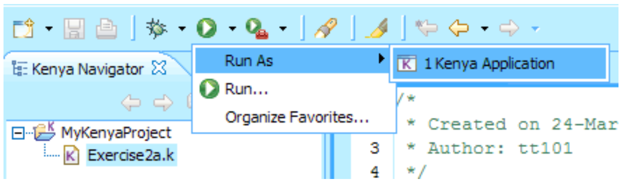
1. Make sure you are working within the **Kenya Perspective**
   Only the Kenya perspective has the functionality to launch Kenya programs.
2. **Open** the code you want to run or **select** it in the **Kenya Explorer**
3. You can now use the **launch shortcuts** in either the toolbar or the right click menu of the selected file in the Explorer:



A further way to launch or debug is the **Run** menu item, but this is not further explained here, you can find out yourself.

## Running

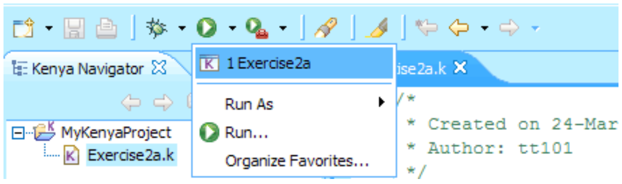
To create a new launch configuration, simply click on the arrow next to the icon (in toolbar) and select **Run As** -> **Kenya Application**:



```
/*
 * Created on 24-Mar
 * Author: tt101
 */
```

The application will be launched using the external java executable. Note that this can fail if the executable is not on your system PATH. You can try this by opening a command line prompt or shell and typing 'java' (without quotes). *KenyaEclipse* uses the same java executable.

Once you have run an application, you will find a shortcut to run it in the same menu:

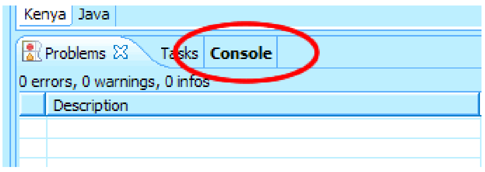


```
/*
 * Created on 24-Mar
 * Author: tt101
 */
```

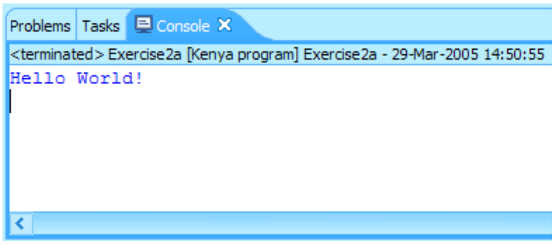
The menu shows the most recent runs with the most recent at the top. Clicking on the **icon itself** will actually launch the **most recent one**.

<u>What happens when we run the thing?</u>

You will notice that something happens in the background and that shortly afterwards a new view shows up at the bottom:



The bold print on the Console tab means that the view contains 'new contents'. In this case they are of course the results of our run:



```
<terminated> Exercise2a [Kenya program] Exercise2a - 29-Mar-2005 14:50:55
Hello World!
```

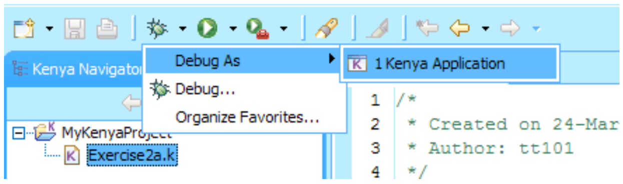
Congratulations, you have run your first application using *KenyaEclipse*!

## Debugging

Debugging allows us to inspect each and every instruction that the program executes, so that we can see where it goes wrong. (Duh, **de-bugg**ing)

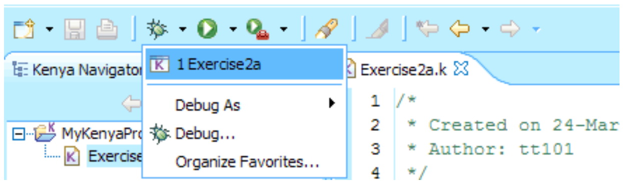
**Make sure** you have read the previous chapter on **running your code**, as it explains some common initial steps.

Debugging is almost exactly the same as running, simply use the arrow next to the bug icon instead:



```
/*
 * Created on 24-Mar
 * Author: tt101
 */
```

Since we executed our code earlier, you will find that a shortcut to our program already exists, although it would appear after we first debug as well:



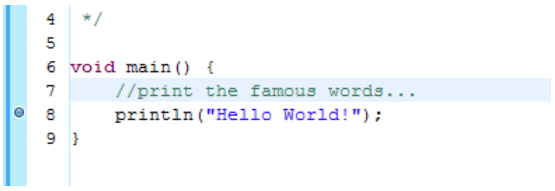
```
/*
 * Created on 24-Mar
 * Author: tt101
 */
```

Again, when we run, the Console view will display the output (it looks exactly the same as above).

## Breakpoints - what was that about tracing instructions?

To be able to **trace**, you will first need to create what is called a **breakpoint**. This is really easy and can be quite powerful. It tells the debugger where to suspend the execution to allow you to inspect the current variables and other things.

The way it works in KenyaEclipse is that you simply **double click** on the **Annotation Ruler** in the editor, next to the line in which you create the breakpoint. This will create a breakpoint on the left most instruction in the line and place an icon on the ruler so that you know it is there:

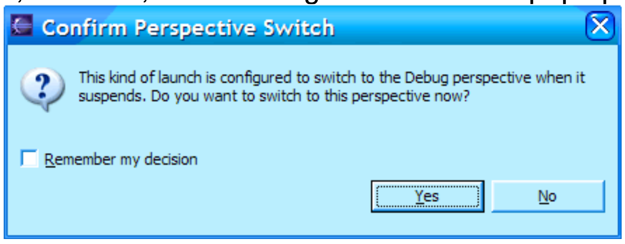


```
 */

void main() {
    //print the famous words...
    println("Hello World!");
}
```

- To remove the breakpoint you can double click on it again and it will disappear.
- Right clicking will allow you to disable/enable it temporarily.
- It is of course possible to set finer grained breakpoints. This is done by **first selecting** the instruction you wish to break on and then double clicking on the ruler.
- To check which instruction is the one used for the breakpoint, you can single click on the breakpoint annotation.
- To change it, just remove the breakpoint and make a different one using the steps above.

## The Debug Perspective

Click on **debug** again, this time, the following window should pop up:



Confirm Perspective Switch

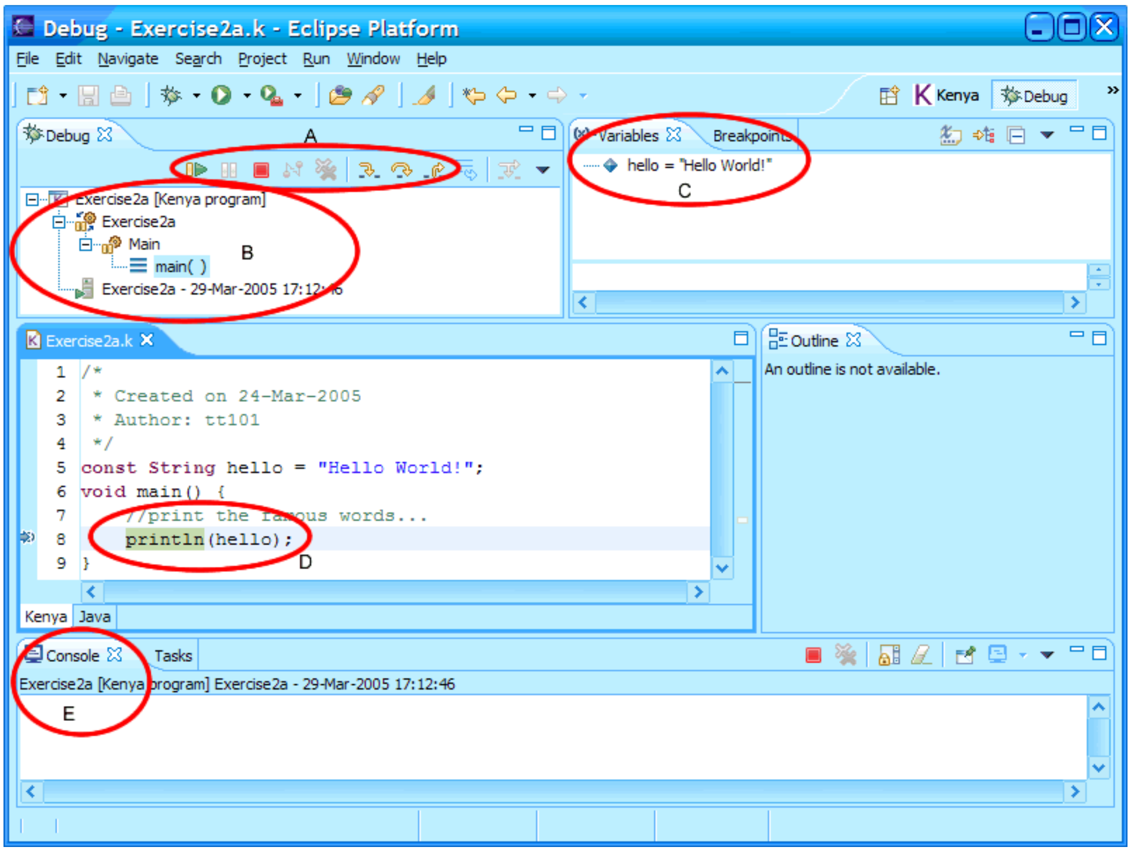
This kind of launch is configured to switch to the Debug perspective when it suspends. Do you want to switch to this perspective now?

☐ Remember my decision

Yes No

The debugger operates in its own perspective and provides appropriate functionality. To switch to the Debug Perspective, you can click on **Yes** in the posed dialog.

After confirming the dialog, you will see roughly this picture:


Debug - Exercise2a.k - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Kenya
Debug
Debug
A
Variables
Breakpoints
C
B
Exercise2a.k
D
Outline
An outline is not available.
Kenya
Java
Console
Tasks
E

```
Exercise2a [Kenya program]
  Exercise2a
    Main
      main( )
  Exercise2a - 29-Mar-2005 17:12:46
```

```
hello = "Hello World!"
```

```
/*
 * Created on 24-Mar-2005
 * Author: tt101
 */
const String hello = "Hello World!";
void main() {
    //print the famous words...
    println(hello);
}
```

```
Exercise2a [Kenya program] Exercise2a - 29-Mar-2005 17:12:46
```

Let me first explain the five components that are highlighted above:

A These tools control the execution of the program. They are in order, **resume**, suspend, **stop**, disconnect, clear terminated, **step into**, **step over**, **step return** (and drop to frame). I will soon explain the details for the bold items.

B This view displays the current state of execution down to the method which is currently at the top of the stackframe. If multiple methods are on the stack (one method called the other), then you can click on each one to view its own state of execution in components C and D.

C The variable view allows you to see in detail what variables are declared at the current scope level and watch their values change as you debug your code. The breakpoint view lets you manage your breakpoints for this debug session.

D After clicking on a method in B, this view will jump to the line which is current in that method. In this case you should notice that the execution is supended exactly where we set our breakpoint earlier.

E The console view is back! This view is exactly the same as before and is explained in more detail in a later chapter.

## Resume, Stop, Step

These are the main functions used to control execution of the program in debug mode.
Once it has supended (because of a breakpoint you set), you can resume execution by clicking the **resume** button.
The **stop** button will cause the program to terminate forcefully and immediately. You can use this to cancel the whole process.
**Step into** is used to execute the next instruction (the one highlighted in D).
**Step over** is used to skip the next entire line of instructions (including method calls).
**Step return** is used to execute everything up to the end of the selected method.

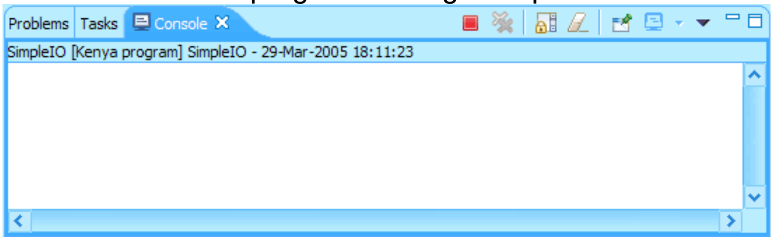
## The Console View

Knowing how to use this view is quite important when working with I/O (Input/Output).
If you are working with I/O, you will have come across the **read** and **print** methods that are built into Kenya.
Let's examine the following code:

```
/*
 * Created on 25-Mar-2005
 * Author: tt101
 */

void main() {
      while(!isEOF()) {
            String input = readString();
            println("You typed: "+input);
      }
}
```

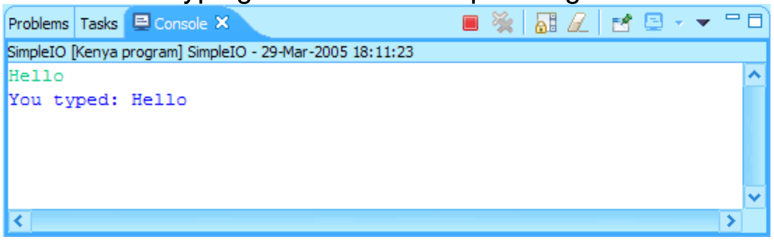
We follow the same steps as above to start execution and following will happen:

The program waiting for input:



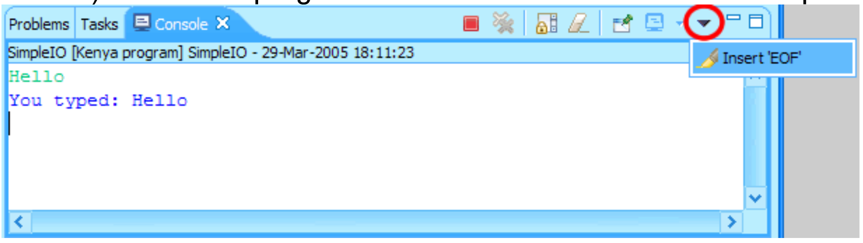
```
SimpleIO [Kenya program] SimpleIO - 29-Mar-2005 18:11:23
```

After typing 'Hello' and then pressing **Enter**:



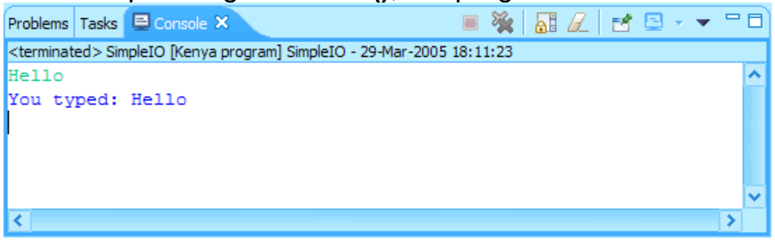
```
SimpleIO [Kenya program] SimpleIO - 29-Mar-2005 18:11:23
Hello
You typed: Hello
```

We can of course type things forever, but at some point we need to end the input using **EOF** (End Of File) so that the program terminates. **Click** the downward pointing arrow:



```
SimpleIO [Kenya program] SimpleIO - 29-Mar-2005 18:11:23
Hello
You typed: Hello
```

Because of the loop testing for **isEOF()**, the program is now able to terminate:



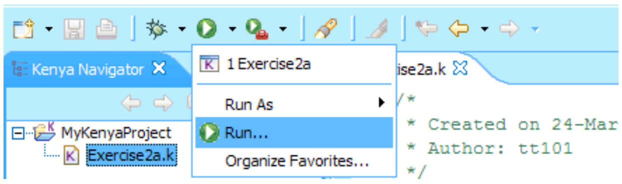
```
<terminated> SimpleIO [Kenya program] SimpleIO - 29-Mar-2005 18:11:23
Hello
You typed: Hello
```

Important points about the console view:

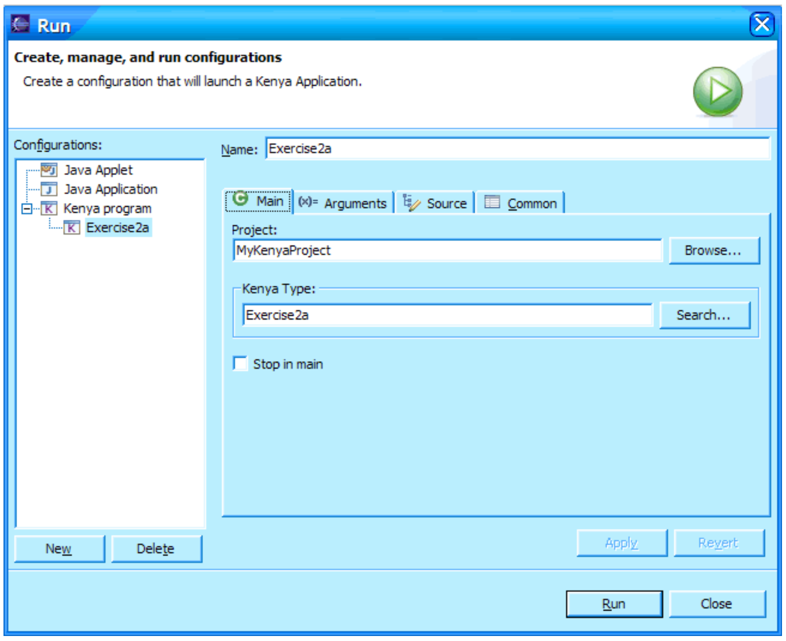
- Input appears in `green` colour
- Output appears in `blue` colour
- Errors appear in `red` colour (hopefully that never happens of course)
- EOF can be achieved by using the console menu
- The console can terminate the program by using the red **stop** button

## Changing launch settings and specifying arguments

If we go back to the beginning, where we launched our program using the launch shortcuts… Try using the run (or debug) menu item instead:



You will be presented with the Launch Configuration Dialog:



Create, manage, and run configurations

Create a configuration that will launch a Kenya Application.

Name: Exercise2a

Project: MyKenyaProject

Kenya Type: Exercise2a

☐ Stop in main

It is best if you explore this dialog yourself. Two options are of particular interest, the first one being the **Arguments** tab, the second one being the **Stop in main** checkbox on the Main tab.

You can supply **runtime arguments** to your program by using the Arguments tab. Arguments are separated by any amount of whitespace, but you can use double quotes to enter arguments that contain spaces.

The **Stop in main** option is only relevant when you debug your program. If the box is checked, the debugger will automatically suspend upon entering the main method, regardless whether breakpoints are set or not.

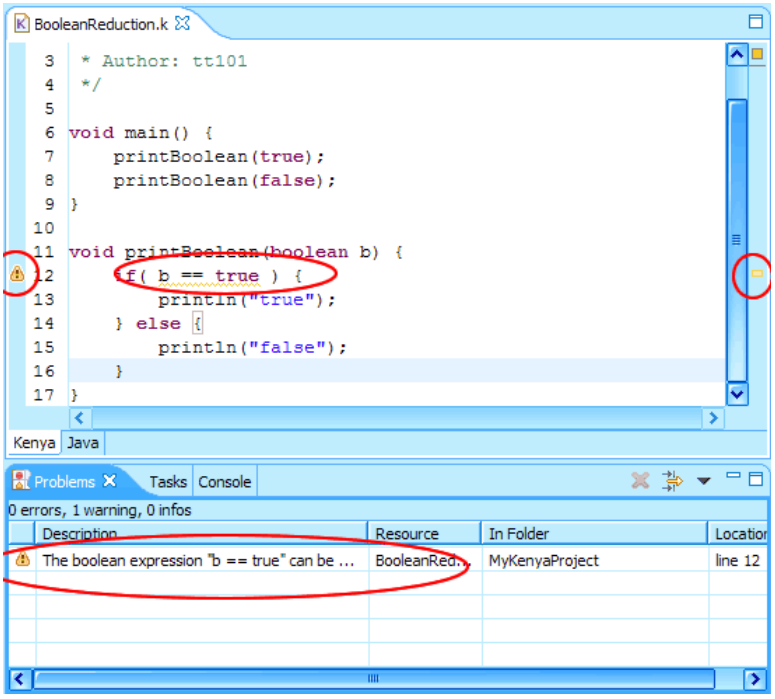
# VI Bad Style – the Style Guidance Module

Style is important in programming and writing ‘bad’ code (code that doesn’t work correctly, is hard to read, is hard to maintain, etc.) either **costs** you **marks**, **time** or your **nerves**.
KenyaEclipse contains a Style guidance module that picks up common mistakes (everybody does it at least once) and helps you to correct them.

Consider the following code snippet:

```
void printBoolean(boolean b) {
  if( b == true ) {
    println("true");
  } else {
    println("false");
  }
}
```

Even if we don't know what b is, we can say that the expression ‘(b == true)’ is exactly the same as the expression ‘b’.
KenyaEclipse will detect this kind of thing for you and help you do the boolean reduction.

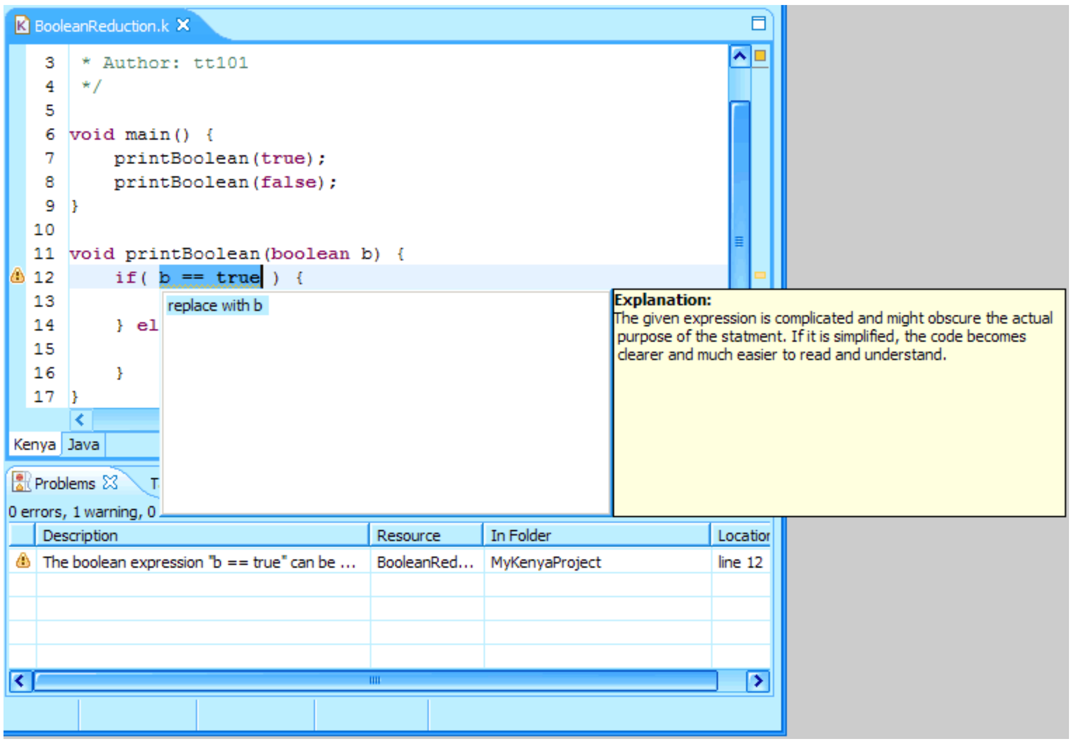


```
 3  * Author: tt101
 4  */
 5
 6 void main() {
 7     printBoolean(true);
 8     printBoolean(false);
 9 }
10
11 void printBoolean(boolean b) {
12     if( b == true ) {
13         println("true");
14     } else {
15         println("false");
16     }
17 }
```

0 errors, 1 warning, 0 infos

| Description | Resource | In Folder | Location |
|---|---|---|---|
| The boolean expression "b == true" can be ... | BooleanRed... | MyKenyaProject | line 12 |

You can see that the style warnings show up in exactly the same way as compile errors do. However, while KenyaEclipse does not offer assistance with compile errors, there is assistance for stilistic mistakes, such as this one. You can trigger the assistant by clicking on the **warning annotation** or by using the **Edit** menu item and selecting **Quick Fix**.



```
 * Author: tt101
 */

void main() {
    printBoolean(true);
    printBoolean(false);
}

void printBoolean(boolean b) {
    if( b == true ) {

    } el

    }
}
```

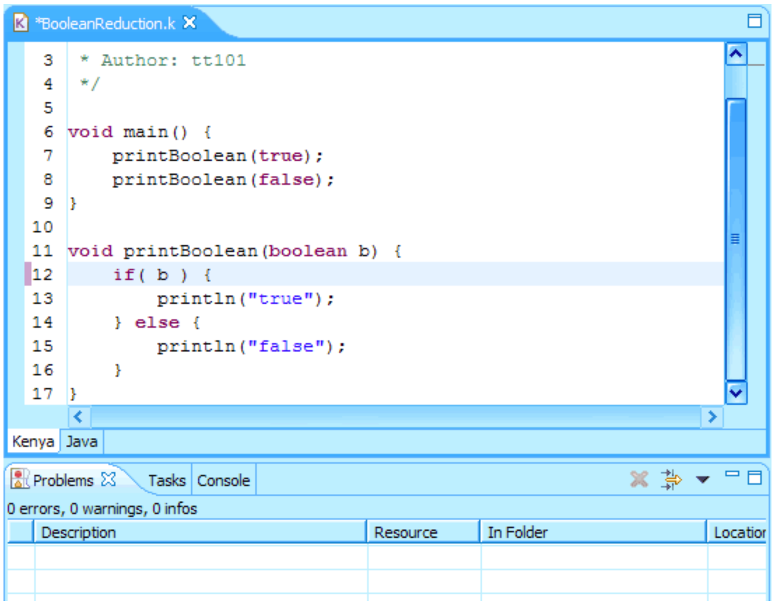
replace with b

**Explanation:**
The given expression is complicated and might obscure the actual purpose of the statment. If it is simplified, the code becomes clearer and much easier to read and understand.

0 errors, 1 warning, 0

| Description | Resource | In Folder | Location |
|---|---|---|---|
| The boolean expression "b == true" can be ... | BooleanRed... | MyKenyaProject | line 12 |

If an automatic solution is available, you will be presented with alternatives for solving the problem at hand and a comprehensive explanation as to why the fix should be applied (shows up after about a second). You can confirm by using **enter** or **double clicking** on the (a) solution you wish to apply. To cancel the dialog, use **ESC**.



```
 * Author: tt101
 */

void main() {
    printBoolean(true);
    printBoolean(false);
}

void printBoolean(boolean b) {
    if( b ) {
        println("true");
    } else {
        println("false");
    }
}
```

0 errors, 0 warnings, 0 infos

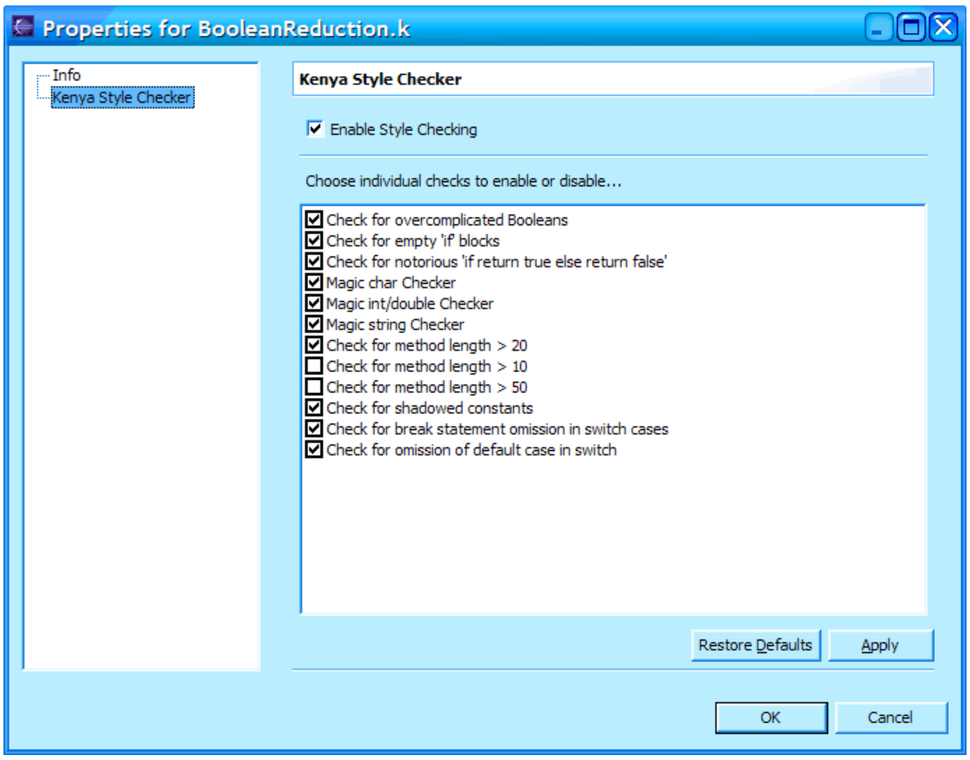
| Description | Resource | In Folder | Location |
|---|---|---|---|

After applying the solution, the document is marked as ‘dirty’, the warning has disappeared and the change ruler contains the change, as if you had done it yourself.

**All changes** applied by the StyleChecker are **un- and re-doable** using the normal undo/redo actions.

## Configuring the module

The StyleChecker is separately configurable for each file. By **right clicking** on the file in question (in the **Kenya Explorer**) and choosing **Properties**, you can reach the configuration page.


Properties for BooleanReduction.k

Info
Kenya Style Checker

**Kenya Style Checker**

[✓] Enable Style Checking

Choose individual checks to enable or disable...

- [✓] Check for overcomplicated Booleans
- [✓] Check for empty 'if' blocks
- [✓] Check for notorious 'if return true else return false'
- [✓] Magic char Checker
- [✓] Magic int/double Checker
- [✓] Magic string Checker
- [✓] Check for method length > 20
- [ ] Check for method length > 10
- [ ] Check for method length > 50
- [✓] Check for shadowed constants
- [✓] Check for break statement omission in switch cases
- [✓] Check for omission of default case in switch

This page allows you to enable or disable style checking altogether or to enable or disable individual checks. If, for example, you disable the ‘Check for overcomplicated Booleans’, you will notice that the warning in our previous bit of code soon disappears (next time the code is compiled).

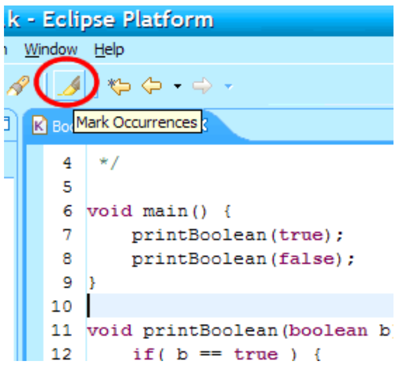
# VII Advanced Editor features

This chapter will explain the following advanced features that are part of KenyaEclipse:

- Variable and type occurrence highlighting
- Code assistance (auto completion)
- Parameter hints for method calls
- Source editing, including
    - Comment toggling
    - Block comment toggling
    - Indentation correction
- Refactoring, including
    - Variable renaming
    - Type (class) renaming

## Occurrence Highlighting

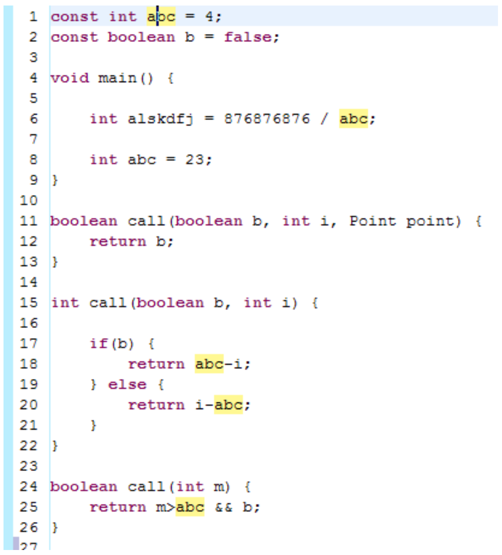
The occurrence highlighter can be enabled by using its button in the toolbar:



```
 */

void main() {
    printBoolean(true);
    printBoolean(false);
}

void printBoolean(boolean b
    if( b == true ) {
```

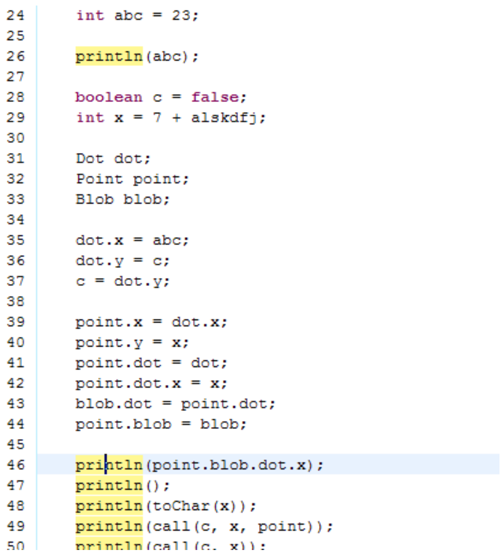
The highlighter is capable of highlighting various things:

Variable references:



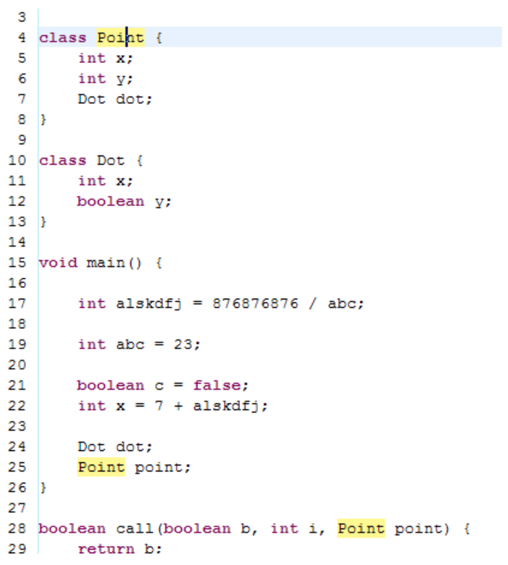
```
const int abc = 4;
const boolean b = false;

void main() {

    int alskdfj = 876876876 / abc;

    int abc = 23;
}

boolean call(boolean b, int i, Point point) {
    return b;
}

int call(boolean b, int i) {

    if(b) {
        return abc-i;
    } else {
        return i-abc;
    }
}

boolean call(int m) {
    return m>abc && b;
}
```

References to methods:



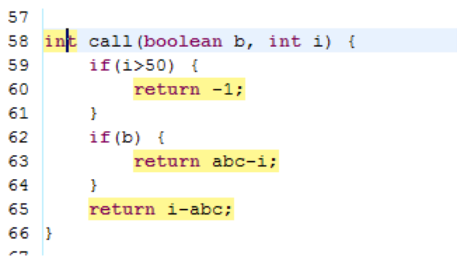
```
    int abc = 23;

    println(abc);

    boolean c = false;
    int x = 7 + alskdfj;

    Dot dot;
    Point point;
    Blob blob;

    dot.x = abc;
    dot.y = c;
    c = dot.y;

    point.x = dot.x;
    point.y = x;
    point.dot = dot;
    point.dot.x = x;
    blob.dot = point.dot;
    point.blob = blob;

    println(point.blob.dot.x);
    println();
    println(toChar(x));
    println(call(c, x, point));
    println(call(c, x));
```

Type (class) references:



```
class Point {
    int x;
    int y;
    Dot dot;
}

class Dot {
    int x;
    boolean y;
}

void main() {

    int alskdfj = 876876876 / abc;

    int abc = 23;

    boolean c = false;
    int x = 7 + alskdfj;

    Dot dot;
    Point point;
}

boolean call(boolean b, int i, Point point) {
    return b;
```

Method exit points:

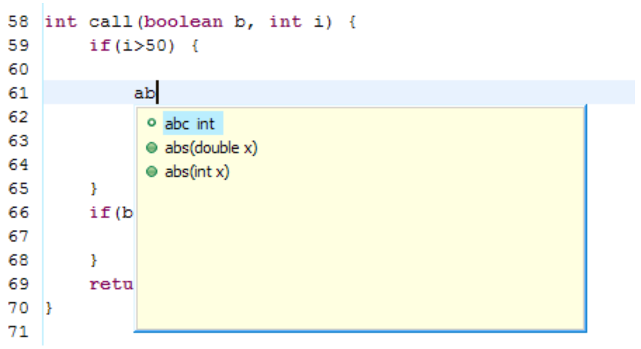


```
int call(boolean b, int i) {
    if(i>50) {
        return -1;
    }
    if(b) {
        return abc-i;
    }
    return i-abc;
}
```

**Note** that Java 1.5 enumerated types will either not be highlighted or that some occurrences may be missed out.

## Code assistance (auto completion)

Code assistance helps you write code faster by letting you complete variable or method names automatically. This function is automatically triggered if you use a qualified name to access fields inside a class, once you type the '.' (dot). You can also bring up the completion dialog yourself by using **Content Assist** in the **Edit** menu (Ctrl+Space on most platforms).
The completion dialog gives choices of all variables and methods that you can use at the current position in your code. While you type more characters, it will narrow down the choices to match.



```
int call(boolean b, int i) {
    if(i>50) {

        ab



    }
    if(b

    }
    retu
}
```

Hitting **enter** or **double clicking** the desired completion will insert the completion into the code.

**Note** that this feature does not support Java 1.5 enumerated types and does not show type parameters for Java 1.5 parameterised methods.

## Parameter Hints

Whether it is a predefined method or one that you wrote yourself, you may not be able to remember the exact order in which you need to enter its arguments. Parameter hints solve this problem by showing you the order, names and types of a method's arguments. This feature can be triggered by selecting **Parameter Hints** in the **Edit** menu.

### Source editing

To speed up common tasks such as commenting (or uncommenting), the **source** menu, which is available through the **right click** menu in the editor contains a few very powerful tools for source editing. The following methods currently exist:

- *Toggle comment* – adds or removes single line commenting (//) for current selection or cursor position
- *Add block comment* – requires multiple characters to be selected, encloses them in a block comment (/* */)
- *Remove block comment* – removes surrounding block comment from current selection or cursor position
- *Correct indentation* – corrects the indentation of the selection according to bracketing rules

### Refactoring

Refactoring is defined to be source tranformation without changing the semantics of the code. There are many useful refactorings for Java. These refactorings are available through the **right click** menu in the editor. The following refactorings currently exist:

- **Rename** – Renames all occurrences of the currently selected (or at cursor position) variable, method or class. You will be asked to specify a new name. This can be useful for changing the name of a variable to something more meaningful (rather than a single-character name for instance).

# VIII Known Issues - Troubleshooting

- *Debugger sometimes does not seem to suspend after using run or one of the step methods.*
  Workaround: click onto the stackframe again (maybe even multiple times). Also try selecting other frames and then returning to the top one.
- *Debugger leaves highlighted bits after terminating*
  Try enabling occurence highlighting using the icon in the toolbar, select a variable name and wait for the highlighting to appear, then disable it again. Alternatively, close the editor and open it again.
- *Debugger doesn't suspend on a breakpoint*
  If this problem persists, try using the 'stop in main' checkbox in the debug configuration (click on the arrow next to the debug button in the toolbar and select 'Debug')
- *Occurrence highlighter does not highlight anything*
  *The occurrence highlighter may not work (or turn itself off) under the following circumstances:*
  - *errored code*
  - *you are using Java 1.5 features*
  - *you have highlighted an overloaded method, where two different declarations have the same number of parameters and return type*
- *Kenya run configuration has no image*
  This seems to be die to an Eclipse bug. If you are using a version below 3.0.2, try upgrading to that version (or higher).
- *There are random exceptions and an error message box pops up*
  Please ask your supervisor for help if the problem persists

# E TIME PLANS

## 9.1 Integration of Kenya with Eclipse

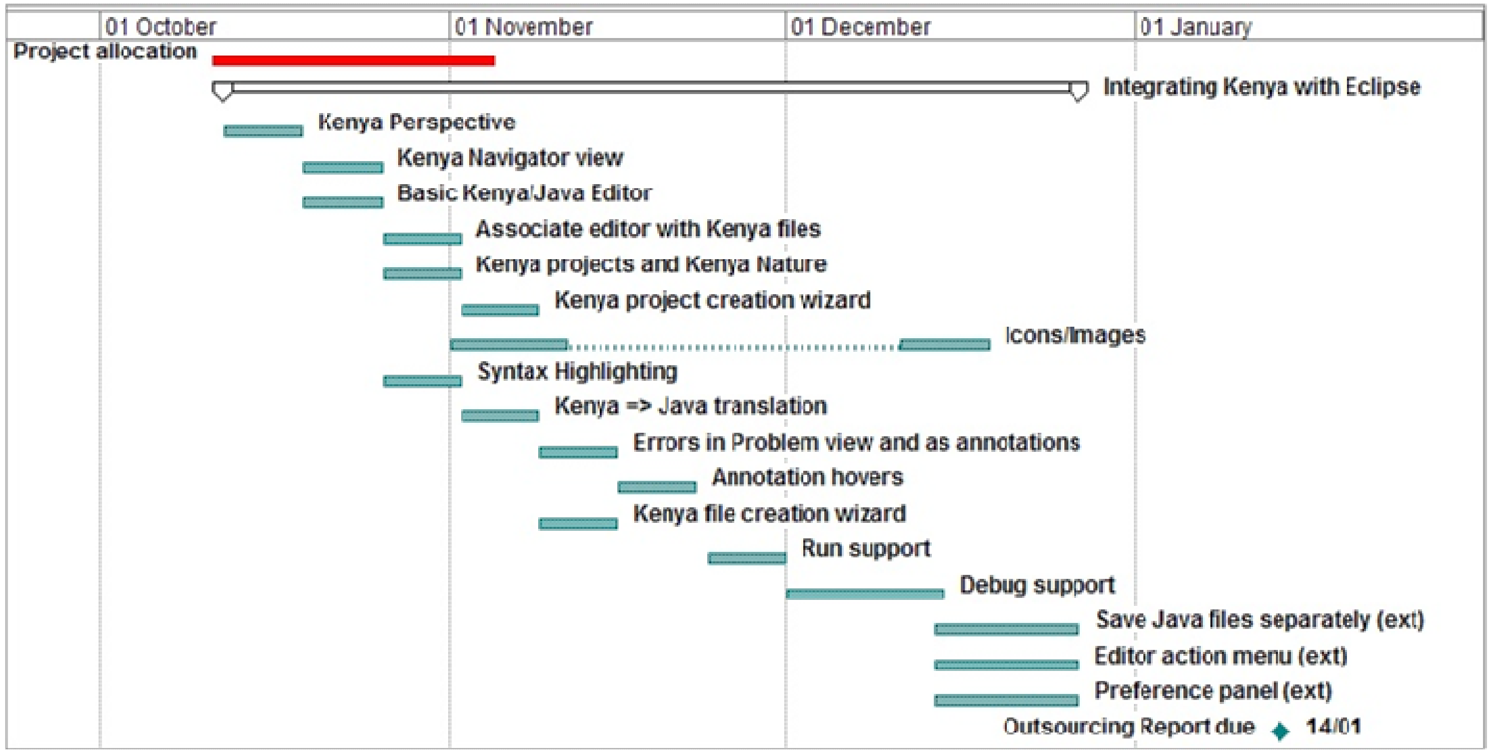


*Figure 9.1.1 - Implementation plan for part I*

While the ‘Editor action panel’ was dropped entirely, implementing Run and Debug support stretched out far into January due to various implementation issues (see section 5.1.2).

## 9.2 KenyaEclipse Feature extension

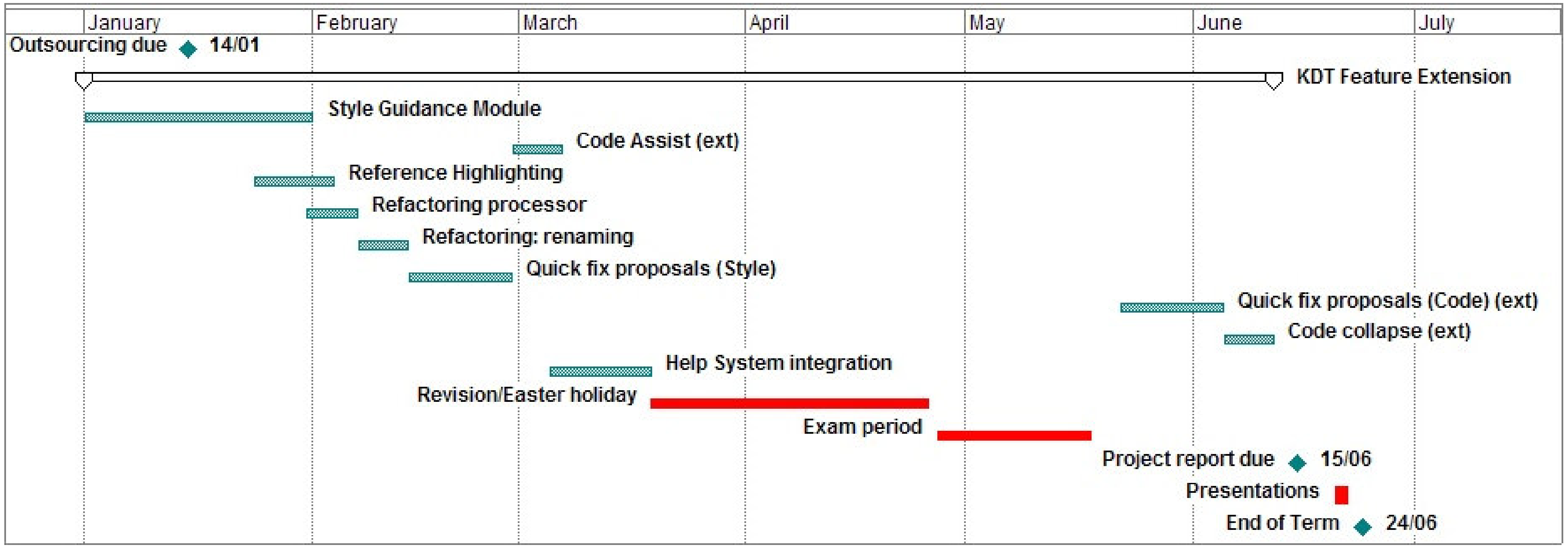


*Figure 9.2.1 - Implementation plan for part II*

Since the schedule had already been pushed back by issues with the Debugger component, Help System integration was postponed until after the exam period. Because of the difficulties that would be associated with attempting to introduce Quick fix proposals for compilation errors (briefly mentioned in section 6.3.2), this feature was dropped entirely and thus the timing was not an issue.

# F LEARNING TO PROGRAM IN ECLIPSE