RAMBUTAN MANUAL

The RAMBUTAN Manual

RAMBUTAN is a literate programming system for Java with TEX, closely resembling `CWEB` and the original `WEB` system.* I developed it using Norman Ramsey's Spidery `WEB`.

This manual is an example of a RAMBUTAN literate program; that is to say, the file `Manual.w` consists of code and documentation written together in the RAMBUTAN idiom. From this common source, the RAMBUTAN system does two things:

```
javatangle Manual
```

extracts a compilable Java applet to compute the first N primes, and

```
javaweave Manual
```

produces a TEX file laying out code and documentation together, including these words.

Actually, the above is a slight oversimplification: `Manual.w` *could* have contained the whole source, but in fact I have distributed the source between `Manual.w`, `Primes.w`, and `Manual.ch`, in order to illustrate multiple source files—but more on that later.

The example code follows this preamble, and introduces the main ideas of literate programming, as relevant to RAMBUTAN. (The reader is assumed to be reasonably familiar with Java and TEX.) After the program there are short explanations of all of RAMBUTAN's features. The important features are few and simple and explained first; the arcana for literate-programming experts come later. A brief annotated bibliography concludes.

Version 1.32 (Dec 2007)

* In other words, what you would expect to be called `JavaWEB`. But since `JavaWEB` sounds too much like a Sun trademark and is a clumsy word anyway, the system as a whole is called RAMBUTAN. But inside RAMBUTAN the usual naming conventions apply: the preprocessors are called `javatangle` and `javaweave` and the TEX macro file is called `javaweb.tex`. (A rambutan, by the way, is a delicious fruit, not unlike a lychee, widely enjoyed in Java and elsewhere.)

1. Computing primes. This is a Java applet that takes two numbers *N1*, *N2* and prints out the *N1*-th prime to the *N2*-th prime.

Like all literate programs, this one consists of a series of numbered **sections**. We are currently in section 1. (Any material before section 1 is called **limbo**; in this case, the introduction.) Most sections consist of a short **text** part followed by a short **code** part. Some sections (such as this one) contain only text, some others contain only code.

Section 1 is always a **starred section**. That just means it has a title: 'Computing primes' in this case. The title is supposed to describe a large group of consecutive sections, and gets printed at the start and on the page headline. Long programs have many starred sections, which behave like chapter headings.

The source for this section begins

```
@* Computing primes.  This is...
```

In the source, `@*` begins a starred section, and any text up to the first period makes up the title.

2. This is an ordinary (i.e., unstarred) section, and its source begins

```
@ This is an ordinary...
```

In the source, `@` followed by space or tab or newline begins an ordinary section.

In the next section things get more interesting.

3. ⟨ Imported packages 3 ⟩ ≡

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
```

This code is used in section 5.

4. The source for section 3 begins

```
@ @<Imported packages@>=
  import java.applet...
```

The result is to make `@<Imported packages@>` an abbreviation for four Java statements—note the `=` in the source.

The bit ⟨ Imported packages 3 ⟩ is called the **section name**, not to be confused with the title of a starred section. Notice how RAMBUTAN has attached the number 3 and inserted a forward reference to section 5.

5. Now we have a whole Java class in abbreviated form. The section ⟨ Imported packages 3 ⟩ is used here, as promised; so are other sections that haven't been defined yet.

`(Primes.java 5)` ≡

```
⟨ Imported packages 3 ⟩
public class Primes extends Applet implements ActionListener
  { ⟨ Fields 12 ⟩
    ⟨ Code for initializing 13 ⟩
    ⟨ The event handler 18 ⟩
  }
```

6. The source for section 5 is

```
@ Now we have...
@(Primes.java@>=
  @<Imported packages@>
  public class Primes extends Applet
    implements ActionListener
    { @<Fields@>
      @<Code for initializing@>
      @<The event handler@>
    }
```

Note the left parenthesis in `(Primes.java 5)`, in contrast with the angle brackets used for other section names. The source for the section name `(Primes.java 5)` is

```
@(Primes.java@>=
```

rather than

```
@<Primes.java@>=
```

Because of this, section 5 is an **output section**: its expansion is output to the specified file `Primes.java`.

Aside: The filename `Primes.java` has to be given by the programmer; RAMBUTAN is not smart enough to figure out the correct filename from context.

7. That's it for the really essential features of a literate programming system: `javatangle` collects the code fragments into a compilable program and `javaweave` cross-references the sections. The remaining features of RAMBUTAN are basically refinements. This example will illustrate a few more features, but the full list can wait till the next chapter of this manual. Meanwhile we'll get on with explaining the program.

8. The algorithmic job of this program is to produce a list of primes, which it does inductively.

First, note that testing p for primeness is easy if we know all the primes $< p$. We set $pmul[j]$ to consecutive odd multiples of $prime[j]$ and check whether we ever hit p. It is enough to try multiples of primes $\le \sqrt{p}$.

```
⟨ Set factor ← true if p is a multiple of a prime 8 ⟩ ≡
  for (int j ← 2; psqr[j] ≤ p; j++)
    { while (pmul[j] < p) pmul[j] +← 2 * prime[j];
      if (pmul[j] ≡ p) factor ← true;
    }
```

This code is used in section 9.

9. Now suppose we have found $prime[1]$ through $prime[k-1]$. We then try successive odd numbers $p > prime[k-1]$ until we find a prime p.

```
⟨ Compute prime[k] 9 ⟩ ≡
  if (k ≡ 1) prime[k] ← 2;
  else if (k ≡ 2) prime[k] ← 3;
  else
    for (int p ← prime[k − 1] + 2; ; p +← 2)
      { boolean factor ← false;
        ⟨ Set factor ← true if p is a multiple of a prime 8 ⟩
        if (¬factor)
          { prime[k] ← p; break;
          }
      }
  pmul[k] ← prime[k]; psqr[k] ← prime[k] * prime[k];
```

This code is used in section 20.

10. ⟨ Arrays for computing primes 10 ⟩ ≡
```
int[] prime ← new int[N2 + 1];
int[] pmul ← new int[N2 + 1]; int[] psqr ← new int[N2 + 1];
```
This code is used in section 20.

11. When we use the code from section 8 in section 9, the source actually gives the section name as

```
@<Set |factor=true| if...@>=
```

with the three dots. Once a section name has appeared in the source RAMBUTAN can complete it from this kind of three-dot shorthand. (And by the way, RAMBUTAN sensibly collapses extra spaces or newlines in section names.)

Another feature is the usage `|factor=true|` which tells `javaweave` to typeset the enclosed text in code-style.

12. The rest of this program is the GUI. Here are the elements for it. (We restrict ourselves to Java 1.1, which more people's browsers will interpret than Java 2.)

The code here includes some comments; literate programs usually need comparatively few comments. RAMBUTAN knows about the `//` comment syntax in Java but not about `/*...*/` comments.

If you need to include strings in the `.java` file that RAMBUTAN can't parse, enclose them in `@=...@>`. A `@=/** javadoc comment */@>` can be inserted in this way.

⟨ Fields 12 ⟩ ≡
```
int N1 ← 0, N2 ← 0; TextField N1_txt, N2_txt; Button run; Panel panel;    ... for input
TextArea disp;    ... for output
```
This code is used in section 5.

13. This method makes a labelled *TextField* and attaches it to *panel*.

⟨ Code for initializing 13 ⟩ ≡
```
TextField new_tf(String str, int n)
  { Panel p ← new Panel(); TextField t ← new TextField(n);
    p.add(new Label(str, Label.CENTER)); p.add(t); p.add(new Label("␣", Label.CENTER));
    panel.add(p);
    return t;
  }
```
See also section 15.

This code is used in section 5.

14. Section 15 has the same section name as section 13. When two or more sections have the same name, RAMBUTAN automatically concatenates them. Note the forward reference in section 13 and the continuation mark '+ ≡ ' in section 15.

15. The applet's *init*() method. Because *disp* here is `Center` in a *BorderLayout*, it will take up any spare space.

⟨ Code for initializing 13 ⟩+ ≡
```
public void init()
  { panel ← new Panel(); N1_txt ← new_tf("N1", 4); N2_txt ← new_tf("N2", 4);
    run ← new Button("run"); panel.add(run); run.addActionListener(this);
    disp ← new TextArea(); disp.setEditable(false);
    setLayout(new BorderLayout()); add("North", panel); add("Center", disp);
  }
```

16. Some (very rare) sections have a **definitions** part.

define *intN*(*i*) ≡ *Integer*.*parseInt*(*N*@& *i*@& *_txt*.*getText*())

17. In section 16 we have a macro. The @& removes any space between its neighbors in the `java` file. Accordingly, *intN*(1) will do something with the variable *N1_txt*, and so on.

18. ⟨ The event handler 18 ⟩ ≡

```
public void actionPerformed(ActionEvent event)
  { run.setEnabled(false);
    try
      { int n1 ← intN(1); int n2 ← intN(2);
        if (n1 ≥ 1 ∧ n2 ≥ n1)
          { N1 ← n1; N2 ← n2;
            ⟨ Compute and display primes 20 ⟩
          }
        else
          { ⟨ Restore old values of N1, N2 19 ⟩
          }
      }
    catch (NumberFormatException ex)
      { ⟨ Restore old values of N1, N2 19 ⟩
      }
    run.setEnabled(true);
  }
```

This code is used in section 5.

19. ⟨ Restore old values of *N1*, *N2* 19 ⟩ ≡

```
if (N1 ≡ 0)
  { N1_txt.setText(""); N1_txt.setText("");
  }
else
  { N1_txt.setText(Integer.toString(N1)); N2_txt.setText(Integer.toString(N2));
  }
```

This code is used in section 18.

20. ⟨ Compute and display primes 20 ⟩ ≡

```
⟨ If too extravagant return 21† ⟩
StringBuffer lyne ← new StringBuffer(); disp.setText("");
⟨ Arrays for computing primes 10 ⟩
for (int k ← 1; k ≤ N2; k++)
  { ⟨ Compute prime[k] 9 ⟩
    String num ← new String(Integer.toString(prime[k]) + "␣");
    if (k ≥ N1)
      { lyne.append(num);
        if (lyne.length() < 64) disp.append(num);
        else
          { disp.append("\n" + num); lyne ← new StringBuffer(num);
          }
      }
  }
```

This code is used in section 18.

21†. ⟨ If too extravagant **return** 21† ⟩ ≡

```
if (N2 − N1 ≥ 2000)
  { disp.setText("Printing␣more␣than␣"); disp.append("2000␣primes␣");
    disp.append("is␣too␣boring\n"); disp.append("Try␣increasing␣N1");
    run.setEnabled(true);
    return;
  }
```

This code is used in section 20.

22†. The source of this program is actually in the file `Primes.w`, while `Manual.w` says

```
@i Primes.w
```

to include that file.

If you look in `Primes.w`, you will find that it considers printing > 1000 primes as already too boring, rather than > 2000 primes. The relevant lines of code have been overridden by the **change file** `Manual.ch`. This last file contains

```
@x
  if (N2-N1 >= 1000)
    { disp.setText("Printing more than ");
    { disp.setText("1000 primes ");
@y
  if (N2-N1 >= 2000)
    { disp.setText("Printing more than ");
    { disp.setText("2000 primes ");
@z
```

and continues with a similar construction containing this section. The section numbers 21 and 22 have daggers attached to indicate that a change file is involved.

A change file consists of constructions of the type

```
@x
⟨Lines quoted from the source file⟩
@y
⟨Replacement lines⟩
@z
```

The change-file name is an optional second input parameter on the command line. Thus

```
javatangle Manual.w Manual.ch
```

or simply

```
javatangle Manual Manual
```

and similarly for `javaweave`.

23. Control codes. Following are the complete set of control codes understood by RAMBUTAN. Only the first two sections are really important.

24. *Basic controls.* These cover the essentials of a literate programming system.

`@`⟨*space*⟩	Begins a new section. (A tab or newline is also read as *space* here.)
`@*`⟨*group title*⟩`.`	Begins a starred section.
`@<`⟨*section name*⟩`@>=`	Section definition, which is really the code-part definition. A section can have at most one such definition. The code can be continued in later sections (see examples in sections 13 and 15).
`@<`⟨*section name*⟩`@>`	Code-part of the named section used. A section can have any number of these. After a section name has first appeared (whether as definition or use) it can be abbreviated using three trailing dots. (See example in section 11).
`@(`⟨*filename*⟩`@>=`	Output-section definition. Written to the named file.
`@u`	Output-section; the filename is inferred by replacing the main source file's extension with `java`.

25. *File controls.* These are for using multiple files

`@i` ⟨*filename*⟩	Includes the file. Must be followed by a newline.
`@x`⟨...⟩`@y`⟨...⟩`@z`	Valid only in change files. The control codes `@x`, `@y`, `@z`, must appear at the beginning of a line, and the rest of such a line is ignored. Any material outside the blocks `@x`⟨...⟩`@y`⟨...⟩`@z` is also ignored.

26. *Special tangle controls.* These are for getting special effects in the output `java` file. We have met the first three in the prime-numbers example.

`@d` ⟨*name*⟩ `=` ⟨*defn*⟩	Defines a macro. [`@D` is equivalent.]
⟨*token1*⟩ `@&` ⟨*token2*⟩	`javatangle` outputs the two tokens without intervening space.
`@=`⟨*code text*⟩`@>`	`javatangle` passes the ⟨*code text*⟩ verbatim into the `java` file.
`@'`⟨*digits*⟩	An octal constant (must be positive). For example, `@'100` tangles to `64` and weaves to ´*100*.
`@"`⟨*digits*⟩	A hexadecimal constant. For example, `@"D0D0` tangles to `53456` and weaves to ″`D0D0`.

27. *Special weave controls.* These are for fine-tuning the typesetting. We have met the first one in the prime-numbers example.

`|`⟨*code fragment*⟩`|`	Used in text, or section names, to format a code fragment in code-style. The ⟨*code fragment*⟩ must not contain section names. [This is the only RAMBUTAN control code not involving `@`.]
`@t`⟨*text*⟩`@>`	The ⟨*text*⟩ is put into a TEX `\hbox`. For example, `|size < @t$2^{15}$@>|` produces $size < 2^{15}$. The ⟨*text*⟩ must not contain newlines.
`@f` ⟨*id1*⟩ ⟨*id2*⟩	Format definition; an optional comment enclosed in braces can follow. [`@F` is equivalent.] Makes `javaweave` treat ⟨*id1*⟩ as it currently treats ⟨*id2*⟩. Format definitions appear between the text part and the code part of a section, together with `@d` macros (in any order).
`@/`	Produces a line break. [Should not be used inside expressions.]
`@#`	Like `@/` but adds some extra vertical space.
`@-`	Like `@/` but indents the next line, to show that it is a continuation line.
`@|`	Recommends a line break, but does not force one. [Can be used inside expressions.]
`@+`	Cancels a line break that might otherwise be inserted by `javaweave`.
`@,`	A thin space.
`@;`	Formats code as if there were a semicolon there.
`@@`	`javaweave` outputs a single `@`. This cannot be used inside `@<`⟨*text*⟩`@>` or similar contexts. An alternative is `\AT!` in text.

28. *Index controls.* These are for fine-tuning the index, and ignored by `javatangle`.

`@^`⟨*text*⟩`@>`	The ⟨*text*⟩ will appear in the index in roman type.
`@.`⟨*text*⟩`@>`	The ⟨*text*⟩ will appear in the index in `typewriter` type.
`@:`⟨*text*⟩`@>`	In the index, the TEX file will have `\9{`⟨*text*⟩`}` and the user can define `\9` freely in TEX.
`@!`⟨*token*⟩	In the index entry for ⟨*token*⟩ the section number will be underlined. A reserved word or an identifier of length one will not be indexed except for underlined entries.

29. Other information. The input syntax for `javatangle` is

`javatangle` ⟨*source file*⟩ ⟨*change file*⟩ `-I`⟨*path*⟩

The ⟨*source file*⟩ has default extension `.w` while the optional ⟨*change file*⟩ has default extension `.ch` and the default ⟨*path*⟩ is the current directory.

The input syntax for `javaweave` is similar:

`javaweave` ⟨*source file*⟩ ⟨*change file*⟩ `-x -I`⟨*path*⟩

The additional `-x` option omits the index.

Both programs also implement the `--version` and `--help` options.

30. If you use `pdftex` on the output of `javaweave`, section-number cross-references will be clickable. Using `\LP{`⟨*section number*⟩`}` in text will also give you a clickable link.

31. TEX macros are in `javaweb.tex`, which is based on the original `webmac.tex` but considerably modified and reorganized. The default format is a standalone Plain TEX document, but if you want to use LATEX, or embed within a larger document, minimal changes will be necessary.

32. To get a table of contents (listing the starred sections), put

```
\contents
```

at the very top of the input file. Unlike in `WEB` and `CWEB`, the table of contents comes first. So you will have to run TEX twice to get an up-to-date list.

If you use `pdftex` the contents will also appear as bookmarks.

33. If you are using a change file and want to view only the changed sections, put

```
\let\maybe\iffalse
```

in the source file or the change file, in the limbo part.

Using this option with `pdftex` will generally produce a lot of clickable links to absent sections, but such links will behave sensibly.

34. Bibliography. The basic introductory reference on literate programming in general is Knuth's article:

Literate Programming, in *The Computer Journal* **27**, 97-111 (1984).

which is also reprinted in Knuth's anthology of the same title. (The prime-numbers example in this manual is adapted from the Knuth article.)

For reviews and links on all aspects of literate programming, see Daniel Mall's literate programming web site:

www.literateprogramming.com

Normal Ramsey's Spidery `WEB` (a generator for `tangle` and `weave` programs) is described in:

Literate programming: Weaving a language-independent WEB, Communications of the ACM, **32**, 1051–1055 (1989)

and archived on CTAN. I made a few modifications (such as adding hyperlinks) to the Spidery `WEB` system itself; such modifications are through change files, so Ramsey's original code is untouched. The change files are included in the RAMBUTAN distribution. Ramsey himself now deprecates Spidery `WEB` and favors the simpler `noweb` system:

Literate programming simplified, IEEE Software, **11**, 97–105 (1994)

which is language independent but sacrifices many features, including automatic cross-referencing. See also Ramsey's web site:

www.eecs.harvard.edu/~nr

I use `noweb` too, but I think Spidery `WEB` still has a place.

Finally, the RAMBUTAN distribution is available from

www.qgd.uzh.ch/projects/rambutan/

and is also archived on CTAN.