STR02-J. Specify an appropriate locale when comparing locale-dependent data

Using locale-dependent methods on locale-dependent data can produce unexpected results when the locale is unspecified. Programming language identifiers, protocol keys, and HTML tags are often specified in a particular locale, usually `Locale.ENGLISH`. Running a program in a different locale may result in unexpected program behavior or even allow an attacker to bypass input filters. For these reasons, any program that inspects data generated by a locale-dependent function must specify the locale used to generate that data.

For example, the following program:

```
public class Example {
  public static void main(String[] args) {
    System.out.println("Title".toUpperCase());
  }
}
```

behaves as expected in an English locale:

```
% java Example
TITLE
%
```

However, most languages that use the Latin alphabet associate the letter `I` as the uppercase version of `i`. But Turkish is an exception: it has a dotted `i` whose uppercase version is also dotted () and an undotted whose uppercase version is undotted (`I`). Changing capitalization on most strings in the Turkish locale [API 2006] may produce unexpected results:

```
% java -Duser.language=tr Example
TTLE
%
```

Many programs only use locale-dependent methods for outputting information, such as dates provided that the locale-dependent data is not inspected by the program, and it may safely rely on the default locale.

Noncompliant Code Example (`toUpperCase()`)

Many web apps, such as forum or blogging software, input HTML and then display it. Displaying untrusted HTML can subject a web app to cross-site scripting (XSS) or HTML injection vulnerabilities. Therefore, it is vital that HTML be sanitized before sending it to a web browser.

One common step in sanitization is identifying tags that may contain malicious content. The `<SCRIPT>` tag typically contains JavaScript code that is executed by a client's browser. Consequently, HTML input is commonly filtered for `<SCRIPT>` tags. However, identifying `<SCRIPT>` tags is not as simple as it appears.

In HTML, tags are case-insensitive and consequently can be specified using uppercase, lowercase, or any mixture of cases. This noncompliant code example uses the locale-dependent `String.toUpperCase()` method to convert an HTML tag to uppercase to check it for further processing. The code must ignore `<SCRIPT>` tags, as they indicate code that is to be discarded. Whereas the English locale would convert `"script"` to `"SCRIPT"`, the Turkish locale will convert `"script"` to `"SCRPT"`, and the check will fail to detect the `<SCRIPT>` tag.

```
public static void processTag(String tag) {
  if (tag.toUpperCase().equals("SCRIPT")) {
    return;
  }
  // Process tag
}
```

Compliant Solution (Explicit Locale)

This compliant solution explicitly sets the locale to English to avoid unexpected results:

```
public static void processTag(String tag) {
  if (tag.toUpperCase(Locale.ENGLISH).equals("SCRIPT")) {
    return;
  }
  // Process tag
}
```

Specifying `Locale.ROOT` is a suitable alternative when an English-specific locale would not be appropriate.

Compliant Solution (Default Locale)

This compliant solution sets the default locale to English before performing string comparisons:

```
public static void processTag(String tag) {
  Locale.setDefault(Locale.ENGLISH);

  if (tag.toUpperCase().equals("SCRIPT")) {
    return;
  }
  // Process tag
}
```

Compliant Solution (`String.equalsIgnoreCase()`)

This compliant solution bypasses locales entirely by performing a case-insensitive match. The `String.equalsIgnoreCase()` method creates temporary canonical forms of both strings, which may render them unreadable, but it performs proper comparison without making them dependent on the current locale [Schindler 12].

```
public static void processTag(String tag) {
  if (tag.equalsIgnoreCase("SCRIPT")) {
    return;
  }
  // Process tag
}
```

This solution is compliant because `equalIgnoreCase()` compares two strings, one of which is plain ASCII, and therefore its behavior is well-understood, even if the other string is not plain ASCII. Calling `equalIgnoreCase()` where both strings may not be ASCII is not recommended, simply because `equalIgnoreCase()` may not behave as expected by the developer.

Noncompliant Code Example (`FileReader`)

Java provides classes for handling input and output, which can be based on either bytes or characters. The byte I/O families derive from the `InputStream` and `OutputStream` interfaces and are independent of locale or character encoding. However, the character I/O families derive from `Reader` and `Writer`, and they must convert byte sequences into strings and back, so they rely on a specified character encoding to do their conversion. This encoding is indicated by the `file.encoding` system property, which is part of the current locale. Consequently, a file encoded with one encoding, such as UTF-8, must not be read by a character input method using a different encoding, such as UTF-16.

Programs that read character data (whether directly using a `Reader` or indirectly using some method such as constructing a `String` from a byte array) must be aware of the source of the data. If the encoding of the data is fixed (such as if the data comes from a file resource that is shipped with the program), then that encoding must be specified by the program. Failure to specify the coding enables an attacker to change the encoding to force the program to read the data using the wrong encoding.

This risk does not apply to programs that read data known to be in the encoding specified by the platform running the program. For example, if the program must open a file provided by the user, it is reasonable to rely on the default encoding, expecting that it will be set correctly.

This noncompliant code example reads its own source code and prints it out, prepending each line with a line number. If the program is run with the argument `-Dfile.encoding=UTF16` while its source file is stored as `UTF8`, the program will save garbage in the output file.

```
import java.io.*;

public class PrintMyself {
  private static String inputFile = "PrintMyself.java";
  private static String outputFile = "PrintMyself.txt";

  public static void main(String[] args) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(inputFile));
    PrintWriter writer = new PrintWriter(new FileWriter(outputFile));
    int line = 0;
    while (reader.ready()) {
      line++;
      writer.println(line + ": " + reader.readLine());
    }
    reader.close();
    writer.close();
  }
}
```

Compliant Solution (`Charset`)

In this compliant solution, both the input and output files are explicitly encoded using UTF8. This program behaves correctly regardless of the default encoding.

```
  public static void main(String[] args) throws IOException {
    Charset encoding = Charset.forName("UTF8");
    BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(inputFile), encoding));
    PrintWriter writer = new PrintWriter(new OutputStreamWriter(new FileOutputStream(outputFile), encoding));

    int line = 0;

    /* Rest of code unchanged */
```

Noncompliant Code Example (`Date`)

The concepts of days and years are universal, but the way in which dates are represented varies across cultures and are therefore specific to locales. This noncompliant code example examines the current date and prints one of two messages, depending on whether or not the month is June:

```
import java.util.Date;
import java.text.DateFormat;
import java.util.Locale;

// ...

public static void isJune(Date date) {
  String myString = DateFormat.getDateInstance().format(date);
  System.out.println("The date is " + myString);
  if (myString.startsWith("Jun ")) {
    System.out.println("Enjoy June!");
  } else {
    System.out.println("It's not June.");
  }
}
```

This program behaves as expected on platforms with a US locale:

```
The date is Jun 20, 2014
Enjoy June!
```

but fails on other locales. For example, the output for a German locale (specified by `-Duser.language=de`) is

```
The date is 20.06.2014
It's not June.
```

Compliant Solution (Explicit Locale)

This compliant solution forces the date to be printed in an English format, regardless of the current locale:

```
String myString = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.US).format(rightNow.getTime());
/* Rest of code unchanged */
```

Compliant Solution (Bypass Locale)

This compliant solution checks the date's `MONTH` attribute without formatting it. Although date representations vary by culture, the contents of a `Calendar` date do not. Consequently, this code works in any locale.

```
if (rightNow.get(Calendar.MONTH) == Calendar.JUNE) {
/* Rest of code unchanged */
```

Risk Assessment

Failure to specify the appropriate locale when using locale-dependent methods on local-dependent data without specifying the appropriate locale may result in unexpected behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STR02-J	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)
CodeSonar	5.4p0	**FB.I18N.DM_CONVERT_CASE** **FB.I18N.DM_DEFAULT_ENCODING** **PMD.Design. SimpleDateFormatNeedsLocale** **PMD.Design. UseLocaleWithCaseConversions**	Consider using Locale parameterized version of invoked method Reliance on default encoding Simple date format needs Locale Use Locale with case conversions
Parasoft Jtest	2020.2	**INTER.CCL** **INTER.CTLC**	Use the optional java.util.Locale parameter Do not call 'Character.toLowerCase(char)' or 'Character.toUpperCase(char)' in an internationalized environment
SonarQube	6.7	**S1449**	Locale should be used in String operations

Android Implementation Details

A developer can specify locale on Android using `java.util.Locale`.

Bibliography

[API 2006]	Class `String`
[Seacord 2015]	STR02-J. Specify an appropriate locale when comparing locale-dependent data LiveLesson
[Schindler 12]	The Policeman's Horror: Default Locales, Default Charsets, and Default Timezones