Localization, Pattern Matching, and Formatting

The java.util.Locale Class

- Adapting programs so that they have global awareness of such differences is called internationalization (a.k.a., "in8ln").
- A locale represents a specific geographical, political, or cultural region. Its two most important attributes are language and country.

The java.util.Locale Class

- Certain classes in the Java Standard Library provide locale-sensitive operations. For example, they provide methods to format values that represent dates, currency and numbers according to a specific locale.
- Developing programs that are responsive to a specific locale is called localization.

The java.util.Locale Class

A locale is represented by an instance of the class java.util.Locale. Many locale-sensitive methods require such an instance for their operation. A locale object can be created using the following constructors:

```
Locale(String language)
Locale(String language, String country)
```

The java.util.Locale Class

Selected Language Codes

Language Code	Language
"en"	English
"no"	Norwegian
"fr"	French

Selected Country Codes

Country Code	Country
"US"	United States (US)
"GB"	Great Britain (GB)
"NO"	Norway
"FR"	France

The java.util.Locale Class

Selected Predefined Locales for Languages

Constant	Language
`Locale.ENGLISH`	Locale with English (`new Locale("en","")`)
`Locale.FRENCH`	Locale with French (`new Locale("fr","")`)
`Locale.GERMAN`	Locale with German (`new Locale("de","")`), i.e., Deutsch.

Selected Predefined Locales for Countries

Constant	Country
`Locale.US`	Locale for US (`new Locale("en","US")`)
`Locale.UK`	Locale for United Kingdom/Great Britain (`new Locale("en","GB")`)
`Locale.CANADA_FRENCH`	Locale for Canada with French language (`new Locale("fr","CA")`)

The java.util.Locale Class

The Locale class provides a get and set method to manipulate the default locale:

```
static Locale getDefault()
static void setDefault(Locale newLocale)
```

The java.util.Locale Class

```
String getDisplayCountry()
String getDisplayCountry(Locale inLocale)
```

Returns a name for the locale's country that is appropriate for display to the user, depending on the default locale in the first method or the inLocale argument in the second method.

```
String getDisplayLanguage()
String getDisplayLanguage(Locale inLocale)
```

Returns a name for the locale's language that is appropriate for display to the user, depending on the default locale in the first method or the inLocale argument in the second method.

```
String getDisplayName()
String getDisplayName(Locale inLocale)
```

Returns a name for the locale that is appropriate for display.

The java.util.Locale Class

- A locale is an immutable object, having two sets of get methods to return the display name of the country and the language in the locale.
- The first set returns the display name of the current locale according to the default locale, while the second set returns the display name of the current locale according to the locale specified as argument in the method call.

```
import java.util.Locale;
public class LocalesEverywhere {
  public static void main(String[] args) {
    Locale locDF = Locale.getDefault();
    Locale locNO = new Locale("no", "NO"); // Locale: Norwegian/Norway
    Locale locFR = new Locale("fr", "FR"); // Locale: French/France
    // Display country name for Norwegian locale:
    System.out.println("In " + locDF.getDisplayCountry() + "(default)" +
       ": " + locNO.getDisplayCountry());
    System.out.println("In " + locNO.getDisplayCountry() +
      ": " + locNO.getDisplayCountry(locNO));
    System.out.println("In " + locFR.getDisplayCountry() +
      ": " + locNO.getDisplayCountry(locFR));
    // Display language name for Norwegian locale:
    System.out.println("In " + locDF.getDisplayCountry() +  "(default)" +
       ": " + locNO.getDisplayLanguage());
    System.out.println("In " + locNO.getDisplayCountry() +
      ": " + locNO.getDisplayLanguage(locNO));
    System.out.println("In " + locFR.getDisplayCountry() +
      ": " + locNO.getDisplayLanguage(locFR));
  }
}
```

Example’s result

Output from the program:
In United Kingdom(default): Norway
In Norway: Norge
In France: Norvège
In United Kingdom(default): Norwegian
In Norway: norsk
In France: norvégien

The java.util.Date Class

- The Date class represents time as a long integer which is the number of milliseconds measured from January 1, 1970 00:00:00.000 GMT.
- This starting point is called the epoch. The long value used to represent a point in time comprises both the date and the time of day.

The java.util.Date Class

The Date class provides the following constructors:

```
Date()
Date(long milliseconds)
```

The default constructor returns the current date and time of day. The second constructor returns the date/time corresponding to the specified milliseconds after the epoch.

The Date class has mostly deprecated methods, and provides date operations in terms of milliseconds only. However, it is useful for printing the date value in a standardized long format

The java.util.Date Class

The toString() method (called implicitly in the print statements) prints the date value in a long format. The date value can be manipulated as a long integer, and a *negative* long value can be used to represent a date *before* the epoch.

```
import java.util.Date;
public class UpToDate {
  public static void main(String[] args) {
    // Get the current date:
    Date currentDate = new Date();
    System.out.println("Date formatted: " + currentDate);
    System.out.println("Date value in milliseconds: " + currentDate.getTime());
    // Create a Date object with a specific value of time measured
    // in milliseconds from the epoch:
    Date date1 = new Date(1200000000000L);
    // Change the date in the Date object:
    System.out.println("Date before adjustment: " + date1);
    date1.setTime(date1.getTime() + 1000000000L);
    System.out.println("Date after adjustment: " + date1);
    // Compare two dates:
    String compareStatus = currentDate.after(date1) ? "after" : "before";
    System.out.println(currentDate + " is " + compareStatus + " " + date1);
    // Set a date before epoch:
    date1.setTime(-1200000000000L);
    System.out.println("Date before epoch: " + date1);
  }
}
```

Example’s result

Date formatted: Wed Mar 05 00:37:28 EST 2008

Date value in milliseconds: 1204695448609

Date before adjustment: Thu Jan 10 16:20:00 EST 2008

Date after adjustment: Tue Jan 22 06:06:40 EST 2008

Wed Mar 05 00:37:28 EST 2008 is after Tue Jan 22 06:06:40 EST 2008

Date before epoch: Tue Dec 22 21:40:00 EST 1931

The java.util.Calendar Class

- A calendar represents a specific instant in time that comprises a date and a time of day.
- The abstract class java.util.Calendar provides a rich set of date operations to represent and manipulate many variations on date/time representation.
- However, the locale-sensitive formatting of the calendar is delegated to the DateFormat class

Static Factory Methods to Create a Calendar

```
static Calendar getInstance()
static Calendar getInstance(Locale loc)
```

The first method returns a calendar with the current date/time using the default time zone and default locale.

The second returns a calendar with the current date/time using the default time zone and specified locale.

Interoperability with the Date Class

`Date getTime()`

Returns the date/time of the calendar in a Date object, as an offset in milliseconds from the epoch.

`void setTime(Date date)`

Sets the current calendar's date/time from the value of the specified date.

Selected get and set Methods

Information in a calendar is accessed via a field number. The Calendar class defines field numbers for the various fields (e.g., year, month, day, hour, minutes, seconds) in a calendar.

Selected get and set Methods

Constant	Field Number that indicates:
`Calendar.WEEK_OF_YEAR` `Calendar.WEEK_OF_MONTH`	The week number within the current year and within the current month, respectively.

`Calendar.DAY_OF_YEAR`
`Calendar.DAY_OF_MONTH`
`Calendar.DATE`
`Calendar.DAY_OF_WEEK`
	The day number within the current year, the current month, the current date (same as `DAY_OF_MONTH`) and the current week, respectively.

`Calendar.YEAR`
`Calendar.MONTH`
`Calendar.HOUR`
`Calendar.HOUR_OF_DAY`
`Calendar.MINUTE`
`Calendar.SECOND`
`Calendar.MILLISECOND`
	The year within the calendar.
	The month within the year.
	The hour within the day (12-hour clock).
	The hour within the day (24-hour clock).
	The minutes within the hour.
	The seconds within the minute.
	The milliseconds within the second.

Manipulating a Calendar

```
System.out.println(calendar.getTime());
// Tue Nov 29 10:20:03 EET 2011
calendar.add(Calendar.MONTH, 13);
// Add 13 more months
System.out.println(calendar.getTime());
// Sat Dec 28 10:20:03 EET 2012
```

Comparing Calendars

```
int compareTo(Calendar anotherCalendar)
```

Implements Comparable<Calendar>, thus calendars can be compared (as offsets in milliseconds from the epoch).

The java.text.DateFormat Class

- For dealing with text issues like formatting and parsing dates, time, currency and numbers, the Java Standard Library provides the java.text package.
- The abstract class DateFormat in this package provides methods for formatting and parsing dates and time.

Static Factory Methods to Create a Date/Time Formatter

- The class DateFormat provides formatters for dates, time of day, and combinations of date and time for the default locale or for a specified locale.
- The factory methods provide a high degree of flexibility when it comes to mixing and matching different formatting styles and locales. However, the formatting style and the locale cannot be changed after the formatter is created.
- The factory methods generally return an instance of the concrete class SimpleDateFormat, which is a subclass of DateFormat.

Static Factory Methods to Create a Date/Time Formatter

```
static DateFormat getInstance()
```

Returns a default date/time formatter that uses the DateFormat.SHORT style for

both the date and the time

Static Factory Methods to Create a Date/Time Formatter

```
static DateFormat getDateInstance()
static DateFormat getDateInstance(int dateStyle)
static DateFormat getDateInstance(int dateStyle,
                                  Locale loc)
```

These three methods return a formatter for dates.

Static Factory Methods to Create a Date/Time Formatter

```
static DateFormat getTimeInstance()
static DateFormat getTimeInstance(int timeStyle)
static DateFormat getTimeInstance(int timeStyle,
                                  Locale loc)
```

These three methods return a formatter for time of day.

Static Factory Methods to Create a Date/Time Formatter

```
static DateFormat getDateTimeInstance()
static DateFormat getDateTimeInstance(
                int dateStyle, int timeStyle)
static DateFormat getDateTimeInstance(
        int dateStyle, int timeStyle, Locale loc)
```

The last three methods return a formatter for date and time. The no-argument methods return a formatter in default style(s) and in default locale.

Formatting Styles for Date and Time

Style for Date/Time	Description	Examples (Locale: US)
DateFormat.DEFAULT	Default style pattern.	Mar 6, 2008 6:08:39 PM
DateFormat.FULL	Full style pattern.	Thursday, March 6, 2008 6:08:39 PM EST
DateFormat.LONG	Long style pattern.	March 6, 2008 6:08:39 PM EST
DateFormat.MEDIUM	Medium style pattern.	Mar 6, 2008 6:08:39 PM
DateFormat.SHORT	Short style pattern.	3/6/08 6:08 PM

Formatting Dates

- A date/time formatter can be applied to a Date object by calling the format() method.
- The value of the Date object is formatted according to the formatter used.

```
import java.text.DateFormat;
import java.util.*;
class UsingDateFormat {
  public static void main(String[] args) {
    // Create some date/time formatters:
    DateFormat[] dateTimeFormatters = new DateFormat[] {
        DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL,
                Locale.US),
        DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG,
                Locale.US),
        DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM,
                Locale.US),
        DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.SHORT,
                Locale.US)
    };
String[] styles = { "FULL", "LONG", "MEDIUM", "SHORT" };  // Style names:
Date date = new Date();  // Format current date/time using different date
formatters:
    int i = 0;
    for(DateFormat dtf : dateTimeFormatters)
      System.out.printf("%-6s: %s%n", styles[i++], dtf.format(date));
  }
}
```

Output from the program:

FULL : Tuesday, November 29, 2011 10:48:19 AM EET
LONG : November 29, 2011 10:48:19 AM EET
MEDIUM: Nov 29, 2011 10:48:19 AM
SHORT : 11/29/11 10:08 AM
FULL : 29 Ноябрь 2011 г. 10:48:19 EET
LONG : 29 Ноябрь 2011 г. 10:48:19 EET
MEDIUM: 29.11.2011 0:48:19
SHORT : 29.11.11 0:48

Parsing Strings to Date/Time

Although we have called it a date/time formatter, the instance returned by the factory methods mentioned earlier is also a parser that converts strings into date/time values.

Parsing Strings to Date/Time

Example illustrates the parsing of strings to date/time values. It uses the Norwegian locale defined at (1).

Four locale-specific date formatters are created at (2). Each one is used to format the current date and the resulting string is parsed back to a Date object:

```
String strDate = df.format(date);                    // (4)
Date parsedDate = df.parse(strDate);                 // (5)
```

Parsing Strings to Date/Time

The string is parsed according to the locale associated with the formatter.

Being lenient during parsing means allowing values that are incorrect or incomplete.

Lenient parsing is illustrated at (6):

```
System.out.println("32.01.08|" +
dateFormatters[0].parse("32.01.08|"));
```

Managing the Calendar and the Number Formatter

- Each date/time formatter has a Calendar that is used to produce the date/time values from the Date object. In addition, a formatter has a number formatter (NumberFormat) that is used to format the date/time values.
- The calendar and the number formatter are associated when the date/time formatter is created, but they can also be set programmatically by using the methods shown below.

Managing the Calendar and the Number Formatter

`void setCalendar(Calendar calendar)`

Set the calendar to use for values of date and time. Otherwise, the default calendar for the default or specified locale is used.

`Calendar getCalendar()`

Get the calendar associated with this date/time formatter.

`void setNumberFormat(NumberFormat numberFormatter)`

Set the number formatter to use for values of date and time.

`NumberFormat getNumberFormat()`

Get the number formatter associated with the date/time formatter.

The java.text.NumberFormat Class

- The abstract class NumberFormat provides methods for formatting and parsing numbers and currency values.
- Using a NumberFormat is in many ways similar to using a DateFormat.

Static Factory Methods to Create a Number Formatter

- The NumberFormat class provides factory methods for creating locale-sensitive formatters for numbers and currency values.
- However, the locale cannot be changed after the formatter is created.
- The factory methods return an instance of the concrete class java.text.DecimalFormat or an instance of the final class java.util.Currency for formatting numbers or currency values, respectively.
- Although we have called the instance a formatter, it is also a parser—analogous to using a date/time formatter

Static Factory Methods to Create a Number Formatter

```
static NumberFormat getNumberInstance()
static NumberFormat getNumberInstance(Locale loc)

static NumberFormat getCurrencyInstance()
static NumberFormat getCurrencyInstance(Locale loc)
```

The first two methods return a general formatter for numbers, i.e., a number formatter.

The next two methods return a formatter for currency amounts, i.e., a currency formatter.

Formatting Numbers and Currency

The following code shows how we can create a number formatter for the Norwegian locale and use it to format numbers according to this locale.

Note the grouping of the digits and the decimal sign used in formatting according to this locale.

```
Double num = 12345.6789;
Locale locNOR = new Locale("no", "NO");     // Norway
NumberFormat nfNOR =
                  NumberFormat.getNumberInstance(locNOR);
String formattedNumStr = nfNOR.format(num);
System.out.println(formattedNumStr);       // 12 345,679
```

Formatting Numbers and Currency

The following code shows how we can create a currency formatter for the Norwegian locale, and use it to format currency values according to this locale. Note the currency symbol and the grouping of the digits, with the amount being rounded to two decimal places.

```
NumberFormat cfNOR =
       NumberFormat.getCurrencyInstance(locNOR);
String formattedCurrStr = cfNOR.format(num);
System.out.println(formattedCurrStr);
                                    // kr 12 345,68
```

Formatting Numbers and Currency

```
String format(double d)
String format(long l)
```

Formats the specified number and returns the resulting string.

```
Currency getCurrency()
void setCurrency(Currency currency)
```

The first method returns the currency object used by the formatter. The last method allows the currency symbol to be set explicitly in the currency formatter, according to the ISO 4217 currency codes. For example, we can set the Euro symbol in a fr_France currency formatter with this method.

Parsing Strings to Numbers

A number formatter can be used to parse strings that are textual representations of numeric values. The following code shows the Norwegian number formatter from above being used to parse strings.

```
out.println(nfNOR.parse("9876.598")); // 9876
out.println(nfNOR.parse("9876,598")); // (2) 9876.598
```

Specifying the Number of Digits

```
void setMinimumIntegerDigits(int n)
int getMinimumIntegerDigits()
void setMaximumIntegerDigits(int n)
int getMaximumIntegerDigits()
void setMinimumFractionDigits(int n)
int getMinimumFractionDigits()
void setMaximumFractionDigits(int n)
int getMaximumFractionDigits()
```

Sets or gets the minimum or maximum number of digits to be allowed in the integer or decimal part of a number.

String Pattern Matching Using Regular Expressions

Regular Expression Fundamentals

The simplest form of a pattern is a character or a sequence of characters that matches itself.

The pattern o, comprising the character o, will only match itself in the target string (i.e., the input).

```
Index:   01234567890123456789012345678901234567
Target:  All good things come to those who wait
Pattern: o
Match:
(5,5:o)(6,6:o)(17,17:o)(22,22:o)(26,26:o)(32,32:o)
```

Regular Expression Fundamentals

The characters in the target are read from left to right sequentially and matched against the pattern. A match is announced when the pattern matches a particular occurrence of (zero or more) characters in the target. Six matches were found for the pattern o in the given target.

A match is shown in the following notation:

(start_index,end_index:group)

Regular Expression Fundamentals

The example below searches for the pattern who in the given target, showing that three matches were found:

```
Index:   0123456789012345678901234567890123456789
Target:  Interrogation with who, whose and whom.
Pattern: who
Match:   (19,21:who)(24,26:who)(34,36:who)
```

Regular Expression Fundamentals

The regular expression notation uses a number of metacharacters (\, [], -, ^, $, ., ?, *, +, (), |) to define its constructs, i.e., these characters have a special meaning when used in a regular expression.

A character is often called a non-metacharacter when it is not supposed to have any special meaning.

Characters

- The pattern \t will match a tab character in the input, and the pattern \n will match a newline in the input.
- Since the backslash (\) is a metacharacter, we need to escape it (\\) in order to use it as a non-metacharacter in a pattern.
- Any metacharacter in a pattern can be escaped with a backslash (\).
- Note the similarity with escape sequences in Java strings, which also use the \ character as the escape character

Character Classes

- The notation [] can be used to define a pattern that represents a set of characters, called a character class.
- A ^ character is interpreted as a metacharacter when specified immediately after the [character. In this context, it negates all the characters in the set. Anywhere else in the [] construct, it is a non-metacharacter.
- The pattern [^aeiouAEIOU] represents the set of all characters that excludes all vowels

Character Classes

- The ‘-’ character is used to specify intervals inside the [] notation. If the interval cannot be determined for a ‘-’ character, it is treated as a non-metacharacter.
- For example, in the pattern [-A-Z], the first ‘-’ character is interpreted as a non-metacharacter, but the second occurrence is interpreted as a metacharacter that represents an interval.

Selected Character Classes

Character Classes	Matches
`[abc]`	`a`, `b`, or `c` (*simple class*)
`[^abc]`	Any character except `a`, `b`, or `c` (*negation*)
`[a-zA-Z]`	`a` through `z` or `A` through `Z`, inclusive (*range*)
`[a-d[m-p]]`	`a` through `d`, or `m` through `p`, i.e., `[a-dm-p]` (*union*)
`[a-z&&[def]]`	`d`, `e`, or `f` (*intersection*)
`[a-z&&[^bc]]`	`a` through `z`, except for `b` and `c`, i.e., `[ad-z]` (*subtraction*)
`[a-z&&[^m-p]]`	`a` through `z`, and not `m` through `p`, i.e., `[a-lq-z]` (*subtraction*)

Selected Predefined Character Classes

Pre-defined Character Classes	Matches
.	Any character (may also match a line terminator)
\d	A digit, i.e., [^0-9]
\D	A non-digit, i.e., [^\d]
\s	A whitespace character, i.e., [\t\n\x0B\f\r]
\S	A non-whitespace character, i.e., [^\s]
\w	A word character, i.e., [a-zA-Z_0-9]
\W	A non-word character: [^\w]

Selected Predefined Character Classes

```
01234567890123456789012345678901234567890123456789012345678901234567890
Who is who? Whose is it? To whom it may concern. How are you?
Pattern: .[Hh]o
Match:   (0,2:Who)(7,9:who)(12,14:Who)(28,30:who)(48,50: Ho)

012345678901234567890
01-03-49 786 09-09-09
Pattern: \d\d-\d\d-\d\d
Match:   (0,7:01-03-49)(13,20:09-09-09)
```

Boundary Matchers

Sometimes we are interested in finding a pattern match at either the beginning or the end of a string/line. This can be achieved by using boundary matchers (also called anchors), as shown in Table

Boundary Matcher	**Matches** *(R is a regular expression.)*
^R	*Anchoring* at the beginning of a string/line
R$	*Anchoring* at the end of a string/line

Boundary Matchers

```
Index:   0123456789012345678901234567 8
Target:  Who is who? Who me? Who else?
Pattern: \?$
Match:   (28,28:?)
```

Logical Operators

Selected Logical Operators

Logical Operator	(R and U are regular expressions) Matches	Example
`R | U`	Either `R` or `U`. (*Logical OR*)	`^[a-z]|\?$`, a lowercase letter at the beginning or a ? at the end of the line.
`RU`	`R` followed by `U`. (*Logical AND, concatenation*)	`[Jj][aA][vV][aA]`, any occurrence of `Java` in upper or lowercase letters.
`(R)`	`R` as a *group*.	`(^[a-z])|(\?$)`

Logical Operators

The logical operators are shown in increasing order of precedence, analogous to the logical operators in boolean expressions. Here is an example that uses all three logical operators for recognizing any case-insensitive occurrence of Java or C++ in the input:

```
Index:   0123456789012345678901234567890
Target:  JaVA jAvA C++ jAv c+++1 javan C+
Pattern: ([Jj][aA][vV][aA])|([Cc]\+\+)
Match:
(0,3:JaVA)(5,8:jAvA)(10,12:C++)(18,20:c++)(24,27:java)
```

Quantifiers

Quantifiers are powerful operators that repeatedly try to match a regular expression with the remaining characters in the input. These quantifiers (also called repetition operators) are defined as follows:

- R?, that matches the regular expression R zero or one time.
- R*, that matches the regular expression R zero or more times.
- R+, that matches the regular expression R one or more times.

Quantifiers

The pattern a? is matched with a target string in the following example:

```
Index:   012345
Target:  banana
Pattern: a?
Match:(0,0:)(1,1:a)(2,2:)(3,3:a)(4,4:)(5,5:a)(6,6:)
```

Examples

The pattern \d\d?-\d\d?-\d\d? is used as a simplified date format in the following example.

The regular expression \d\d? represents any one or any two digits.

```
Index:   01234567890123456789012345678901
Target:  01-3-49 23-12 9-09-09 01-01-2010
Pattern: \d\d?-\d\d?-\d\d?
Match:   (0,6:01-3-49)(14,20:9-09-09)(22,29:01-01-20)
```

Examples

The pattern a* is interpreted as a non-zero sequence of a's or as the empty string (meaning no a's). The engine returns an empty string as the match, when the character in the input cannot be a part of a sequence of a's.

```
Index:   01234567
Target:  baananaa
Pattern: a*
Match:
(0,0:)(1,2:aa)(3,3:)(4,4:a)(5,5:)(6,7:aa)(8,8:)
```

Examples

The pattern (0|[1-9]\d*)\.\d\d recognizes all non-zero-leading, positive floating-point numbers that have at least one digit in the integral part and exactly two decimal places.

Note that the regular expression \d* is equivalent to the regular expression [0-9]*.

```
Index:   01234567890123456789012345678890
Target:  .50 1.50 0.50 10.50 00.50 1.555
Pattern: (0|[1-9]\d*)\.\d\d
Match:   (4,7:1.50)(9,12:0.50)(14,18:10.50)
         (21,24:0.50)(26,29:1.55)
```

Examples

The pattern a+ is interpreted as a non-zero sequence of a's, i.e., at least one a. Compare the results below with the results for using the pattern a* above on the same target. No empty strings are returned when an a cannot be matched in the target.

```
Index:   01234567
Target:  baananaa
Pattern: a+
Match:    (1,2:aa)(4,4:a)(6,7:aa)
```

Examples

The regular expression \d+ represents all permutations of digits. The pattern \d+.\d+ represents all positive floating-point numbers that have at least one digit in the integral part and at least one digit in the fraction part.

Note that \d+ is equivalent to [0-9]+.

```
Index:   01234567890123456789012345678
Target:  .50 1.50 0. 10.50 00.50 1.555
Pattern: \d+\.\d+
Match:
(4,7:1.50)(12,16:10.50)(18,22:00.50)(24,28:1.555)
```

Greedy Quantifiers

- The quantifiers presented above are called greedy quantifiers. Such a quantifier reads as much input as possible, and backtracks if necessary, to match as much of the input as possible.
- In other words, it will return the longest possible match.

Example

The example below illustrates greediness. The pattern <.+> is supposed to recognize a tag, i.e., a non-zero sequence of characters enclosed in angle brackets (< >).

The example below shows that only one tag is found in the target. The greedy quantifier + returns the longest possible match in the input.

```
0123456789012345678901234567890123456789012345678901234
My <>very<> <emphasis>greedy</emphasis> regex
Pattern: <.+>
Match:
(3,38:<>very<> <emphasis>greedy</emphasis>)
```

Reluctant (Lazy) Quantifiers

- A reluctant quantifier (also called lazy quantifier) only reads enough of the input to match the pattern. Such a quantifier will apply its regular expression as few times as possible, only expanding the match as the engine backtracks to find a match for the overall regular expression.
- In other words, it will return the shortest possible match.

Reluctant (Lazy) Quantifiers

The example below illustrates reluctantness/laziness. The pattern <.+?> uses the reluctant quantifier +?, and is supposed to recognize a tag as before.

The example below shows the result of applying the pattern to a target.

The reluctant quantifier +? returns the shortest possible match for each tag recognized in the input.

```
0123456789012345678901234567890123456789012345678901234567
My <>very<> <emphasis>reluctant</emphasis> regex
Pattern: <.+?>
Match:
(3,10:<>very<>)(12,21:<emphasis>)(31,41:</emphasis>)
```

Reluctant (Lazy) Quantifiers

- We can improve the matching by using the trick shown in this pattern: <[^>]+>.
- Since the match has two enclosing angle brackets, the pattern negates the end angle bracket, creating a character class that excludes the end angle bracket. The engine can keep expanding the tag name as long as no end angle bracket is found in the input. When this bracket is found in the input, a match can be announced, without incurring the penalty of backtracking.
- Note that the pattern below is using the greedy quantifier +

Example

```
0123456789012345678901234567890123456789012345678
My <>very<> <emphasis>powerful</emphasis> regex
Pattern: <[^>]+>
Match:   (12,21:<emphasis>)(30,40:</emphasis>)
```

Possessive Quantifiers

- Lastly, there are the possessive quantifiers that always consume the entire input, and then go for one make-or-break attempt to find a match.
- A possessive quantifier never backtracks, even if doing so would succeed in finding a match.
- There are certain situations where possessive quantifiers can outperform the other types of quantifiers

Quantifier Classification

Greedy	Reluctant	Possessive	Matches (R *is a regular expression*)
R?	R??	R?+	R zero or one time, i.e., R is optional.
R*	R*?	R*+	R zero or more times.
R+	R+?	R++	R one or more times.

Escaping Metacharacters

- A regular expression can be specified as a string expression in a Java program. In the declaration below, the string literal "who" contains the pattern who.

```
String p1 = "who";  // regex: who
```

- The pattern \d represents a single digit character. If we are not careful in how we specify this pattern in a string literal, we run into trouble.

```
String p2 = "\d";
// Java compiler: Invalid escape sequence!
```

Escaping Metacharacters

- For every backslash in a regular expression, we need to escape it in the string literal, i.e. specify it as a backslash pair (\\).
- This ensures that the Java compiler accepts the string literal, and the string will contain only one backslash for every backslash pair that is specified in the string literal.
- A backslash contained in the string is thus interpreted correctly as a backslash in the regular expression.

```
String p3 = "\\d";   // regex: \d
String p4 = "\\.";   // regex: \.
                        (i.e. the . non-metacharacter)
String p5 = ".";     // regex: .
                        (i.e. the . metacharacter)
```

Escaping Metacharacters

If we want to use a backslash as a non-metacharacter in a regular expression, we have to escape the backslash (\), i.e use the pattern \\.

In order to escape these two backslashes in a string literal, we need to specify two consecutive backslash pairs (\\\\).

Each backslash pair becomes a single backslash inside the string, resulting in the two pairs becoming a single backslash pair, which is interpreted correctly in the regular expression, as the two backslash characters represent a backslash non-metacharacter.

```
String nonMetaBackslash = "\\\\";   // regex: \\
                                (i.e. the \ non-metacharacter)
```

Examples

Each backslash in the regular expression is escaped in the string literal.

```
String p6 = "\\d\\d-\\d\\d-\\d\\d";
// regex: \d\d-\d\d-\d\d
String p7 = "\\d+\\.\\d+";
// regex: \d+\.\d+
String p8 = "(^[a-z])|(\\?$)";
// regex: (^[a-z])|(\?$)
```

The java.util.regex.Pattern Class

The two classes **Pattern** and **Matcher** in the java.util.regex package embody the paradigm for working efficiently with regular expressions in Java.

The java.util.regex.Pattern Class

1. Compiling the regular expression string into a Pattern object which constitutes the compiled representation of the regular expression (i.e., a pattern) mentioned earlier:

```
Pattern pattern = Pattern.compile(regexStr);
```

2. Using the Pattern object to obtain a Matcher (i.e., an engine) for applying the pattern to a specified input of type java.lang.CharSequence:

```
Matcher matcher = pattern.matcher(input);
```

3. Using the operations of the matcher to apply the pattern to the input:

```
boolean eureka = matcher.matches();
```

Compiling a Pattern

The methods below can be used to compile a regular expression string into a pattern and to retrieve the regular expression string from the pattern, respectively.

```
String regexStr = "\\d\\d-\\d\\d-\\d\\d";
// regex: \d\d-\d\d-\d\d
Pattern datePattern = Pattern.compile(regexStr);
```

Compiling a Pattern

`static Pattern compile(String regexStr)`

Compiles the specified regular expression string into a pattern. Throws the unchecked PatternSyntaxException if the regular expression is invalid.

When the source is line-oriented, it is recommended to use the overloaded compile() method that additionally takes the argument Pattern.MULTILINE.

`String pattern()`

Returns the regular expression string from which this pattern was compiled.

Creating a Matcher

The matcher() method returns a Matcher, which is the engine that does the actual pattern matching. This method does not apply the underlying pattern to the specified input. The matcher provides special operations to actually do the pattern matching.

```
Matcher dateMatcher =
datePattern.matcher("01-03-49 786 09-09-09");
```

The Pattern class method ***matches***

- The Pattern class also provides a static convenience method that executes all the steps outlined above for pattern matching. The regular expression string and the input are passed to the static method matches(), which does the pattern matching on the entire input.
- The regular expression string is compiled and the matcher is created each time the method is called.
- *Calling the matches() method is not recommended if the pattern is to be used multiple times.*

```
boolean dateFound =
Pattern.matches("\\d\\d-\\d\\d-\\d\\d", "01-03-49");
// true
```

The Pattern class methods

```
Matcher matcher(CharSequence input)
```

Creates a matcher that will match the specified input against this pattern.

```
static boolean matches(
    String regexStr,
    CharSequence input)
```

Compiles the specified regular expression string and attempts to match the specified input against it. The method only returns true if the entire input matches the pattern.

Splitting

- The normal mode of pattern matching is to find matches for the pattern in the input.
- In other words, the result of pattern matching is the sequences of characters (i.e., the matches, also called groups) that match the pattern.
- Splitting returns sequences of characters that do not match the pattern.
- In other words, the matches are spliced out and the sequences of non-matching characters thus formed from the input are returned in an array of type String.

Splitting

The pattern is used as a delimiter to tokenize the input. The token in this case is a sequence of non-matching characters, possibly the empty string.

The classes StringTokenizer and Scanner in the java.util package also provide the functionality for tokenizing text-based input.

Splitting

The example below shows the results from splitting an input on a given pattern. The input is a ‘|’-separated list of names. The regular expression string is "\\|", where the metacharacter | is escaped in order to use it as a non-metacharacter.

Splitting the given input according to the specified regular expression, results in the array of String shown below.

```
Input: "tom|dick|harry"  Split: "\\|"
Results: { "tom", "dick", "harry" }
```

The split() method can be called on a pattern to create an array by splitting the input according to the pattern.

Splitting

Each successful application of the pattern, meaning each match of the pattern delimiter in the input, results in a split of the input, with the non-matched characters before the match resulting in a new element in the array, and any remaining input being returned as the last element of the array.

```
String[] split(CharSequence input, int limit)
```

Splits the specified input around matches of this pattern. The limit determines how many times this pattern will be applied to the input to create the array.

Implications of the Limit Value in the split() Method

Limit `n`	No. of Applications	Array Length	Other Remarks
`n > 0`	At the most `n-1` times, meaning it can also be fewer if the input was exhausted	No greater than `n`, meaning it can also be smaller if the input was exhausted	Any remaining input is returned in the last element of the array
`n == 0`	As many times as possible to split the entire input	Any length required to split the entire input	Trailing empty strings are discarded
`n < 0`	As many times as possible to split the entire input	Any length required to split the entire input	Trailing empty strings are *not* discarded

The java.util.regex.Matcher Class

- A Matcher is an engine that performs match operations on a character sequence by interpreting a Pattern.
- A matcher is created from a pattern by invoking the Pattern.matcher() method.
- Here we will explore the following three modes of operation for a matcher

1. One-Shot Matching

Using the matches() method in the Matcher class to match

the entire input sequence against the pattern.

```
Pattern pattern =
                Pattern.compile("\\d\\d-\\d\\d-\\d\\d");
Matcher matcher = pattern.matcher("01-03-49");
boolean isMatch = matcher.matches();       // true
matcher = pattern.matcher("1-3-49");
isMatch = matcher.matches();               // false
```

1. One-Shot Matching

The convenience method matches() in the Pattern class in the last subsection calls the matches() method in the Matcher class implicitly.

`boolean matches()`

Attempts to match the entire input sequence against the pattern. The method returns true only if the entire input matches the pattern.

2. Successive Matching

Using the find() method in the Matcher class to successively apply the pattern on the input sequence to look for the next match

Successive Matching

The main steps of successive matching using a matcher are somewhat analogous to using an iterator to traverse a collection. These steps are embodied in the code below.

…

```
Pattern pattern = Pattern.compile(regexStr);
Matcher matcher = pattern.matcher(target);
while(matcher.find()) {
   ...
   String matchedStr = matcher.group();
   ...
}
```

…

Successive Matching

- Once a matcher has been obtained, the find() method of the Matcher class can be used to find the next match in the input (called target in the code).
- The find() returns true if a match was found.
- If the previous call to the find() method returned true, and the matcher has not been reset since then, the next call to the find() method will advance the search in the target for the next match from the first character not matched by the previous match.
- If the previous call to the find() returned false, no match was found, and the entire input has been exhausted.

Successive Matching

`boolean find()`

Attempts to find the next match in the input that matches the pattern. The first call to this method, or a call to this method after the matcher is reset, always starts the search for a match at the beginning of the input.

`String group()`

Returns the characters (substring) in the input that comprise the previous match.

`int start()`

`int end()`

The first method returns the start index of the previous match. The second method returns the index of the last character matched, plus one. The values returned by these two methods define a substring in the input.

Successive Matching

`Matcher reset()`

`Matcher reset(CharSequence input)`

The method resets this matcher, so that the next call to the find() method will begin the search for a match from the start of the current input. The second method resets this matcher, so that the next call to the find() method will begin the search for a match from the start of the new input.

`Matcher usePattern(Pattern newPattern)`

Replaces the pattern used by this matcher with another pattern. This change does not affect the search position in the input.

`Pattern pattern()`

Returns the pattern that is interpreted by this matcher.

3. Match-and-Replace Mode

Using the matcher to find matches in the input sequence and replace them.

In this mode, the matcher allows the matched characters in the input to be replaced with new ones. Details of the methods used for this purpose are given below.

Match-and-Replace Mode

The find() and the appendReplacement() methods comprise the match-and-replace loop, with the appendReplacement() method completing the operation when the loop finishes.

Note that these methods use a StringBuffer, and have not been updated to work with a StringBuilder.

Match-and-Replace Mode

```
Matcher appendReplacement(
          StringBuffer sb,
          String replacement)
```

Implements a non-terminal append-and-replace step, i.e., it successively adds the non-matched characters in the input, followed by the replacement of the match, to the string buffer.

The find() method and the appendReplacement() method are used in lockstep to successively replace all matches, and the appendTail() method is called as the last step to complete the match-and-replace operation.

```
StringBuffer appendTail(StringBuffer sb)
```

Implements a terminal append-and-replace step, i.e., it copies the remaining characters from the input to the string buffer, which is then returned.

It should be called after appendReplacement() operations have completed

Match-and-Replace Mode

`String replaceAll(String replacement)`
Replaces every subsequence of the input that matches the pattern with the specified replacement string. The method resets the matcher first and returns the result after the replacement.

`String replaceFirst(String replacement)`
Replaces the first subsequence of the input that matches the pattern with the specified replacement string. The method resets the matcher first and returns the result after the replacement.

The java.util.Scanner Class

A scanner reads characters from a source and converts them into tokens. The source is usually a text-based input stream containing formatted data. The formatted values in the source are separated by delimiters, usually whitespace.

A token is a sequence of characters in the source that comprises a formatted value. A scanner generally uses regular expressions to recognize tokens in the source input.

The java.util.Scanner Class

A point to note is that a scanner can also use regular expressions to recognize delimiters, which are normally discarded. Such a scanner is also called a tokenizer (also called a lexical analyzer), and the process is called tokenization.

Some scanners also convert the tokens into values of appropriate types for further processing. Scanners with this additional functionality are usually called parsers.

The java.util.Scanner Class

We will discuss two modes of operation for a scanner:

- Tokenizing Mode, for tokenizing a stream of formatted data values.
- Multi-Line Mode, for searching or finding matches in line-oriented input.

Constructing a Scanner

A scanner must be constructed and associated with a source before it can be used to parse text-based data. The source of a scanner is passed as an argument in the appropriate constructor call. Once a source is associated with a scanner it cannot be changed.

```
Scanner(SourceType  source)
```

Returns an appropriate scanner. SourceType can be a String, a File, an InputStream, a ReadableByteChannel, or a Readable (implemented by various Readers).

Lookahead Methods

The Scanner class provides two overloaded hasNext() methods that accept a regular expression specified as a string expression or as a Pattern, respectively. The next token is matched against this pattern.

All primitive types and string literals have a pre-defined format which is used by the appropriate lookahead method.

All lookahead methods return true if the match with the next token is successful.

Lookahead Methods

```
boolean hasNext()
boolean hasNext(Pattern pattern)
boolean hasNext(String pattern)
```

The first method returns true if this scanner has another (string) token in its input.

The last two methods return true if the next token matches the specified pattern or the pattern constructed from the specified string, respectively.

Lookahead Methods

```
boolean hasNextIntegralType ()
boolean hasNextIntegralType(int radix)
```

Returns true if the next token in this scanner's input can be interpreted as a value of the integral type corresponding to IntegralType in the default or specified radix.

The name IntegralType can be Byte, Short, Int or Long, corresponding to the primitive types byte, short, int, or long, respectively.

```
boolean hasNextFPType()
```

Returns true if the next token in this scanner's input can be interpreted as a value of the floating-point type corresponding to FPType.

The name FPType can be Float or Double, corresponding to the types float or double, respectively.

Lookahead Methods

`boolean hasNextBoolean()`

Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false".

`boolean hasNextLine()`

Returns true if there is another line in the input of this scanner

The java.util.Scanner class

A scanner uses white space as its default delimiter pattern to identify tokens. The useDelimiters() method of the Scanner class can be used to set a different delimiter pattern for the scanner during parsing.

Note that a scanner uses regular expressions for two purposes: a delimiter pattern to identify delimiter characters and a token pattern to find a token in the input.

A scanner is able to read and parse any value that has been formatted by a printf method, provided the same locale is used. The useLocale() method of the Scanner class can be used to change the locale used by a scanner.

The java.util.Scanner class

```
Pattern delimiter()
Scanner useDelimiter(Pattern pattern)
Scanner useDelimiter(String pattern)
```

The first method returns the pattern this scanner is currently using to match delimiters. The last two methods set its delimiting pattern to the specified pattern or to the pattern constructed from the specified pattern string, respectively.

```
Locale locale()
Scanner useLocale(Locale locale)
```

These methods return this scanner's locale or set its locale to the specified locale, respectively.

```
int radix()
Scanner useRadix(int radix)
```

These methods return this scanner's default radix or set its radix to the specified radix, respectively

Parsing the Next Token

Corresponding to the hasNext() methods, the Scanner class provides two overloaded next() methods that accept a regular expression as a string expression or as a Pattern, respectively. This pattern is used to find the next token.

Parsing the Next Token

- A call to a parse method first skips over any delimiters at the current position in the source, and then reads characters up to the next delimiter.
- The scanner attempts to match the non-delimiter characters that have been read against the pattern associated with the parse method.
- If the match succeeds, a token has been found, which can be parsed accordingly.
- The current position is advanced to the new delimiter character after the token.
- The upshot of this behavior is that if a parse method is not called when a lookahead method reports there is a token, the scanner will not advance in the input.
- In other words, tokenizing will not proceed unless the next token is “cleared.”
- A scanner will throw an InputMismatchException when it cannot parse the input, and the current position will remain unchanged.

Parsing the Next Token

```
String next()
String next(Pattern pattern)
String next(String pattern)
```

The first method scans and returns the next token as a String. The last two methods return the next string in the input that matches the specified pattern or the pattern constructed from the specified string, respectively.

Parsing the Next Token

`ReturnIntegralType nextIntegralType()`

`ReturnIntegralType nextIntegralType(int radix)`

Returns the next token in the input as a value of primitive type corresponding to IntegralType. The name IntegralType can be Byte, Short, Int, or Long, corresponding to the primitive types byte, short, int, or long, respectively. The name ReturnIntegralType is the primitive type corresponding to the name IntegralType.

`ReturnFPType nextFPType()`

Returns the next token in the input as a value of the primitive type corresponding to FPType. The name FPType can be Float or Double, corresponding to the primitive types float or double, respectively. The name ReturnFPType is the primitive type corresponding to the name FPType.

`boolean nextBoolean()`

Returns the next token in the input as a boolean value.

`String nextLine()`

Advances this scanner past the current line and returns the input that was skipped.

Parsing Primitive Values

- To parse such values, we need to know what type of values occur in what order in the input so that an appropriate lookahead and a corresponding parse method can be used. We also need to know what locale was used to format them and which delimiters separate the individual values in the input.
- The order in which the different type of values occur in the input is specified by the vararg parameter tokenTypes, whose element type is the enum type TokenType. A call to the method parse(), such as the one shown below, thus indicates the order, the type and the number of values to expect in the input.

Miscellaneous Scanner Methods

`Scanner skip(Pattern pattern)`
`Scanner skip(String pattern)`

These methods skip input that matches the specified pattern or the pattern constructed from the specified string, respectively, ignoring any delimiters. If no match is found at the current position, no input is skipped and a NoSuchElementException is thrown.

`MatchResult match()`

Returns the match result of the last scanning operation performed by this scanner.

`IOException ioException()`

Returns the IOException last thrown by this scanner's underlying Readable object.

`Scanner reset()`

Resets this scanner to the default state with regard to delimiters, locale, and radix.

`void close()`

Closes this scanner. When a scanner is closed, it will close its input source if the source implements the Closeable interface (implemented by various Channels, InputStreams, Readers)

Multi-Line Mode

If the input is line-oriented, the scanner can be used to perform search in the input one line at a time.

The methods hasNextLine(), findInLine(), and nextLine() form the trinity that implements the multi-line mode of searching the input with a pattern.

Multi-Line Mode

```
String findInLine(Pattern pattern)
String findInLine(String pattern)
```

These methods attempt to find the next occurrence of the specified pattern or the pattern constructed from the specified string, respectively, ignoring any delimiters.

Formatting Values

Overview

The class **java.util.Formatter** provides the core support for formatted text representation of primitive values and objects through its overloaded format() methods:

```
format(String format, Object... args)
format(Locale l, String format, Object... args)
```

Writes a string that is a result of applying the specified format string to the values in the vararg array args.

The resulting string is written to the destination object that is associated with the formatter.

Formatting Values

- The classes java.io.PrintStream and java.io.PrintWriter also provide an overloaded format() method with the same signature for formatted output.
- These streams use an associated Formatter that sends the output to the PrintStream or the PrintWriter, respectively.
- However, the format() method returns the current Formatter, PrintStream, or PrintWriter, respectively, for these classes, allowing method calls to be chained.

Formatting Values

- The String class also provides an analogous format() method, but it is static.
- Unlike the format() method of the classes mentioned earlier, this static method returns the resulting string after formatting the values.
- In addition, the classes PrintStream and PrintWriter provide the following convenience methods:

```
printf(String format, Object... args)
printf(Locale l, String format, Object... args)
```

These methods delegate the formatting to the format() method in the respective classes.

Defining Format Specifiers

The general syntax of a format specifier is as follows:

%[argument_index][flags][width][precision]conversion

Only the special character % and the formatting conversion are not optional.

Defining Format Specifiers

- The occurrence of the character **%** in a format string marks the start of a format specifier, and the associated formatting conversion marks the end of the format specifier.
- A format specifier in the format string is replaced either by the textual representation of the corresponding value or by the specifier's special meaning.
- The compiler does not provide much help regarding the validity of the format specifier.
- Depending on the error in the format specifier, a corresponding exception is thrown at runtime

Defining Format Specifiers

- The optional **argument_index** has the format i$, or it is the < character.
- In the format i$, i is a decimal integer indicating the position of the argument in the vararg array, starting with position 1. The first argument is referenced by 1$, the second by 2$, and so on.
- The < character indicates the same argument that was used in the preceding format specifier in the format string, and cannot therefore occur in the first format specifier.

Defining Format Specifiers

- The optional **flag** is a character that specifies the layout of the output format.
- The optional **width** is a decimal integer indicating the minimum number of characters to be written to the output.
- The optional **precision** has the format .n, where n is a decimal integer and is used usually to restrict the number of characters. The specific behavior depends on the conversion.

Formatting Conversions

Conversion Specification	Conversion Category	Description
'b','B'	general	If the argument *arg* is `null`, the result is `"false"`. If *arg* is a `boolean` or `Boolean`, the result is string returned by `String.valueOf()`. Otherwise, the result is `"true"`.
'h','H'	general	If the argument *arg* is `null`, the result is `"null"`. Otherwise, the result is obtained by invoking `Integer.toHexString(arg.hashCode())`.
's','S'	general	If the argument *arg* is `null`, the result is `"null"`. If *arg* implements `Formattable`, `arg.formatTo()` is invoked. Otherwise, the result is obtained by invoking `arg.toString()`.
'c','C'	character	The result is a Unicode character.
'd'	integral	The result is formatted as a decimal integer.
'o'	integral	The result is formatted as an octal integer.
'x','X'	integral	The result is formatted as a hexadecimal integer.

Formatting Conversions

Conversion Specification	Conversion Category	Description
'e','E'	floating point	The result is formatted as a decimal number in computerized scientific notation.
'f'	floating point	The result is formatted as a decimal number.
'g','G'	floating point	The result is formatted using computerized scientific notation for large exponents and decimal format for small exponents.
'a','A'	floating point	The result is formatted as a hexadecimal floating-point number with a significand and an exponent.
't','T'	date/time	Prefix for date and time conversion characters.
'%'	percent	The result is the character %, i.e., "%%" escapes the % metacharacter.
'n'	line separator	The result is the platform-specific line separator, i.e., "%n"

Flags

Flag	Integrals			Floating-point				Description
	d	o	x X	e E	f	g G	a A	
'-'	*ok*	*ok*	*ok*	*ok*	*ok*	*ok*	*ok*	Left-justified, requires a positive width. (Also *ok* for general, character, and date/time categories.)
'#'	×	*ok*	*ok*	*ok*	*ok*	×	*ok*	Include radix for integrals. Include decimal point for floating-point.
'+'	*ok*	×	×	*ok*	*ok*	*ok*	*ok*	Include the sign.
' '	*ok*	×	×	*ok*	*ok*	*ok*	*ok*	Leading space for positive values.
'0'	*ok*	*ok*	*ok*	*ok*	*ok*	*ok*	*ok*	Zero-padded, requires a positive width.
','	*ok*	×	×	×	*ok*	*ok*	×	Use locale-specific grouping separator.
'('	*ok*	×	×	*ok*	*ok*	*ok*	×	Negative numbers in parentheses.

Selected Time/Date Composition Conversions

Conversion Specification	Description
'R'	Time formatted for the 24-hour clock as "%tH:%tM"
'T'	Time formatted for the 24-hour clock as "%tH:%tM:%tS".
'r'	Time formatted for the 12-hour clock as "%tI:%tM:%tS %tP". The location of the morning or afternoon marker ('%tp') may be locale-dependent.
'D'	Date formatted as "%tm/%td/%ty".
'F'	ISO 8601 complete date formatted as "%tY-%tm-%td".
'c'	Date and time formatted as "%ta %tb %td %tT %tZ %tY", e.g., Tue Mar 04 17:22:37 EST 2008.

Selected Format Exceptions

Format Exception	Meaning
DuplicateFormatFlagsException	Flag used more than once.
FormatFlagsConversionMismatchException	Flag and conversion not compatible.
IllegalFormatConversionException	Type of argument not compatible with the conversion.
IllegalFormatFlagsException	Invalid flag combination.
IllegalFormatPrecisionException	Precision invalid or not permissible.
IllegalFormatWidthException	Width invalid or not permissible.
MissingFormatArgumentException	A conversion has no corresponding argument.
MissingFormatWidthException	A positive width was not specified.
UnknownFormatConversionException	Conversion is unknown.
UnknownFormatFlagsException	A flag is unknown.

Using the format() Method

The destination object of a Formatter, mentioned earlier, can be any one of the following:

- a StringBuilder, by default
- an Appendable, e.g., a String that implements this interface
- a file specified either by its name or by a File object
- a PrintStream, or another OutputStream

Using the format() Method

Various constructors in the Formatter class:

```
Formatter()
Formatter(Locale l)
Formatter(Appendable a)
Formatter(Appendable a, Locale l)
Formatter(File file)
Formatter(File file, String charset)
Formatter(File file, String charset, Locale l)
Formatter(OutputStream os)
Formatter(OutputStream os, String charset)
Formatter(OutputStream os, String charset, Locale l)
Formatter(String fileName)
Formatter(String fileName, String charset)
Formatter(String fileName, String charset, Locale l)
Formatter(PrintStream ps)
```

That’s all