**8.4. Работа со строками в Java. Строки как объекты. Классы *String*, *StringBuffer* и *StringBuilder***

Класс String инкапсулирует действия со строками. Объект типа String — это строка, состоящая из произвольного числа символов (от 0 до 2109). Литерные константы типа String представляют собой последовательности символов, заключенные в двойные кавычки:

"A", "abcd", "abcd", "Мама моет раму", " ".

Это так называемые длинные строки. Внутри литерной строковой константы не разре-шается использовать ряд символов — вместо них применяются управляющие последо-вательности.

Внутри строки допустимы переносы на новую строку. Но литерные константы с таки-ми переносами запрещены, и нужно ставить управляющую последовательность "\n". К сожалению, такой перенос строки не срабатывает при показе текста в большинстве компонентов (JLabel,JButton и др.). Но для компонентов JTextArea такой перенос строки действует. Разрешены пустые строки, не содержащие ни одного символа. Например, если задана строковая переменная

String s;

то после присваивания

s="";

эта переменная будет ссылаться на пустую строку.

Отметим, что существует принципиальная разница между ситуацией, когда строковая переменная не инициализирована (как у нас s сразу после объявления), и ситуацией, когда она ссылается на пустую строку. В первом случае в переменной хранится значе-ние null, во втором — она ссылается на полноценный объект-строку, просто объект хранит пустую последовательность символов. Но во втором случае из этого объекта можно вызывать методы. Например, s.length() возвратит длину строки, равную нулю. А попытка вызвать любой метод (хотя бы ту же функцию s.length()) в случае, когда s содержит значение null, приведет к возбуждению исключения NullPointerException — *см. разд. 8.8*.

В языке Java строковый и символьный типы несовместимы:

"A" — строка из одного символа;

'A' — число с ASCII-кодом символа "A".

Это несколько усложняет совместную работу со строками и символами.

Строки можно складывать.

Если s1 и s2 — строковые литерные константы или переменные, то результатом опера-ции s1+s2 будет строка, являющаяся сцеплением (конкатенацией) строк, хранящихся в s1 и s2.

Например, в результате операции:

String s="Это "+"моя строка";

в переменной s будет храниться строковое значение "Это моя строка".

Для строк разрешен оператор "+=". Для строковых операндов s1 и s2 выражение s1+=s2 эквивалентно выражению s1=s1+s2.

Любая строка (набор символов) является объектом — экземпляром класса String. Пе-ременные типа String — это ссылки на объекты, что следует учитывать при передаче параметров строкового типа в подпрограммы, а также при многократных изменениях строк. При каждом изменении строки в динамической области памяти создается новый объект, а прежний превращается в мусор. Поэтому при многократных изменениях строк в цикле возникает много мусора, что нежелательно.

Очень частая ошибка — попытка сравнения строк с помощью оператора ==.

Например, результатом выполнения листинга 8.10 будет вывод в консольное окно строки:

s1 не равно s2

т. к. объекты-строки имеют в памяти разные адреса

**Листинг 8.10. Проблема сравнения строк на равенство**

String s1="Строка типа String";

String s2="Строка";

s2+=" типа String";

if(s1==s2)

System.out.println("s1 равно s2");

else

System.out.println("s1 не равно s2");

Оператор equals сравнивает строки по содержанию.

Поэтому если бы вместо s1==s2 мы написали s1.equals(s2), то получили бы ответ:

s1 равно s2.

Удивительным может показаться факт, что результатом выполнения листинга 8.11 бу-дет вывод в консольное окно строки:

s1 равно s2.

**Листинг 8.11. Проблема сравнения на равенство литерных констант**

String s1="Строка";

String s2="Строка";

if(s1==s2)

System.out.println("s1 равно s2");

else

System.out.println("s1 не равно s2");

Дело в том, что компилятор Java анализирует имеющиеся в коде программы литерные константы и на этапе компиляции для одинаковых по содержанию констант использует одни и те же объекты-строки. И даже если задать переменную s2 как

String s2 = "Стр"+"ока";

мы получим тот же результат. Но вот если динамически создать новый независимый объект с помощью конструктора

String s2 =new String("Строка");

или, например, прочитать значение s2 из пункта редактирования текста

String s2 = jTextArea1.getText();

мы получим сообщение, что s1 не равно s2.

В остальных случаях то, что строковые переменные ссылочные, обычно никак не влия-ет на работу со строковыми переменными, и с ними можно действовать так, как если бы они содержали сами строки.

В классе String есть много конструкторов, предназначенных для преобразования тек-стовой информации из первоначального формата в объект-строку. Перечислим некото-рые из них.

Пусть:

s1, s2, charsetName и subS имеют тип String;

charArray — массив символов char[];

byteArray — массив символов byte[];

i, index и count (счет, количество) — целочисленные переменные или значения.

Тогда:

String() — создание объекта-строки с пустым содержимым строки. Присваивание s1=new String() эквивалентно присваиванию s="" (за исключением того, что объект с пустой строкой для литерной константы "" хранится в единственном экземпляре, как разъяснялось ранее, а конструктор String() всегда создает новый, отличный от прежних объект с пустым содержимым строки).

String(s1) — создание копии объекта-строки, на которую ссылается строковая пере-менная s1.

String(charArray) — массив символов копируется в объект-строку.

String(charArray,index,count) — то же, что в предыдущем случае, но копируется count символов, начиная с символа, имеющего позицию index. При выходе за границы массива возбуждается исключение IndexOutOfBoundsException — *см. разд. 8.8*.

String(stringBuffer) — копирование последовательности символов объекта-строки stringBuffer, имеющей тип StringBuffer, в объект типа String. Про класс StringBuffer рассказывается далее.

String(stringBuilder) — копирование последовательности символов объекта-строки stringBuilder, имеющей тип StringBuilder, в объект типа String. Про класс StringBuilder рассказывается далее.

String(byteArray) — массив байтов копируется и преобразуется (декодируется) в объект-строку платформо-зависимым образом. При этом считается, что в байтовом массиве находится последовательность символов из набора (charset), установленно-го по умолчанию (default) для данного компьютера. В связи с этим результат деко-дирования будет гарантированно неправильным на компьютерах с другим нацио-нальным набором символов по умолчанию. Поэтому данный вариант конструктора имеет смысл использовать только для дешифровки последовательностей англоязыч-ных символов.

String(byteArray,index,count) — то же, что в предыдущем случае, но декодируется count байт начиная с байта, имеющего позицию index. При выходе за границы масси-ва возбуждается исключение IndexOutOfBoundsException — *см. разд. 8.8*.

String(byteArray,charsetName) — массив байтов копируется и преобразуется (декоди-руется) в объект-строку платформо-независимым образом. При этом считается, что в байтовом массиве находится последовательность символов из набора символов, за-данного с помощью строки charsetName. Например, могут быть использованы имена "ISO-8859-1" (стандартный латинский алфавит в 8-битовой кодировке), "UTF-8", "UTF-16" (символы Unicode) и др. Если указанное имя набора символов не поддерживает-ся, возбуждается исключение UnsupportedEncodingException — *см. разд. 8.8*.

String(byteArray,index,count,charsetName) — то же, что в предыдущем варианте кон-структора, но декодируется не весь байтовый массив, а count байтов начиная с байта, имеющего позицию index. При выходе за границы массива возбуждается исключе-ние IndexOutOfBoundsException — *см. разд. 8.8*.

Кроме конструкторов в классе String имеется ряд других методов класса. Важнейшие из них:

String.valueOf(параметр) — возвращает строку типа String, являющуюся результатом преобразования параметра в строку. Параметр может быть любого примитивного или объектного типа.

String.valueOf(charArray, index, count) — функция, аналогичная предыдущей в слу-чае, когда параметр является массивом символов, но здесь преобразуется count сим-волов, начиная с символа, имеющего индекс index.

В классе String предусмотрен ряд методов объектов, т. е. таких методов, которые могут быть вызваны из объектов типа String. Перечислим важнейшие из них, разбив на не-сколько категорий.

Общие характеристики строки:

s1.length() — возвращает длину строки (число 16-битовых символов Unicode, со-держащихся в строке). Длина пустой строки равна нулю;

s1.isEmpty() — возвращает true в случае, если строка, на которую ссылается пере-менная s1, пустая. Если строка не пустая, возвращается false;

s1.intern() — возвращает ссылку на объект-строку, содержащуюся в хранилище строковых констант и имеющую такую же последовательность символов. Если же такой строки в хранилище не найдено, объект-строка, на который ссылается s1, до-бавляется в хранилище, и возвращается ссылка на нее. Данный метод имеет смысл использовать для экономии памяти только при хранении очень большого количества объектов-строк.

Преобразование строки в массив:

s1.getBytes() — возвращает массив типа byte[], полученный в результате платфор-мо-зависимого преобразования символов строки в последовательность байтов;

s1.getBytes(charsetName) — то же, но с указанием кодировки (charset). Например, мо-гут быть использованы имена "ISO-8859-1", "UTF-8", "UTF-16" (символы Unicode) и др.;

s1.toCharArray() — возвращает массив символов, скопированных из строки s1.

Преобразование регистра строки:

s1.toLowerCase() — возвращает строку с символами, скопированными из строки s1 и преобразованными к нижнему регистру (строчным буквам). Имеется вариант мето-да, делающего такое преобразование с учетом конкретной кодировки (locale);

s1.toUpperCase() — возвращает строку с символами, скопированными из строки s1 и преобразованными к верхнему регистру (заглавным буквам). Имеется вариант мето-да, делающего такое преобразование с учетом конкретной кодировки (locale).

Сравнение строк и их частей:

s1.equals(s2) — возвращает true в случае, когда строка s2 имеет тип String и ее со-держимое (последовательность символов) совпадает с содержимым строки s1. В случае несовпадения последовательностей символов s1 и s2 или если s2 имеет тип, отличный от String, возвращается false;

*324 Глава 8*

При сравнении с буферизованными строками StringBuffer и StringBuilder *(см. далее)* функция всегда возвращает false, даже если содержимое строки s2 совпадает с содер- жимым строки s1. Для сравнения с содержимым строк StringBuffer и StringBuilder необ- ходимо использовать функцию contentEquals.

s1.equalsIgnoreCase(s2) — то же, но при сравнении строк игнорируются различия в регистре символов (строчные и заглавные буквы не различаются);

s1.contentEquals(sb1) — булева функция, проверяющая на совпадение содержимого строки s1 типа String и объекта sb1. В случае если sb1 — строка, имеющая тип String, StringBuffer или StringBuilder *(см. далее)*, и ее содержимое (последователь- ность символов) совпадает с содержимым s1, возвращается true. Иначе возвращает- ся false;

s1.startsWith(subS) — возвращает true в случае, когда строка s1 начинается с симво- лов строки subs;

s1.startsWith(subs,index1) — возвращает true в случае, когда символы строки s1 с позиции index1 начинаются с символов строки subs;

s1.endsWith(subS) — возвращает true в случае, когда строка s1 заканчивается после- довательностью символов, содержащихся в строке subS;

s1.regionMatches(index1,s2,index2,count) — булева функция, сравнивающая подстро- ки двух строк, s1 и s2. Сравнение начинается с позиции index1 для первой строки и с позиции index2 для второй, сравнивается count символов. В случае совпадения по- следовательностей символов возвращается true, иначе false. При недопустимых значениях index1, index2 или count возвращается false;

s1.regionMatches(ignoreCase,index1,s2,index2,count) — то же, но с дополнительным булевым параметром ignoreCase. Если он имеет значение false, то сравнение проис- ходит так же, как в предыдущем методе (регистрочувствительно). Если true — срав- нение идет регистронечувствительно, т. е. заглавные и строчные буквы считаются эквивалентными, и регистр букв не влияет на результат сравнения.

Копирование части строки:

s1.substring(index1) — возвращает строку типа String с символами, скопированными из строки s1, начиная с позиции index1. При выходе за пределы допустимых индексов возбуждается исключительная ситуация IndexOutOfBoundsException — *см. разд. 8.8*;

s1.substring(index1,index2) — возвращает строку типа String с символами, скопиро- ванными из строки s1, начиная с позиции index1 и кончая позицией index2-1. Длина получившеейся строки равна index2-index1. При выходе за пределы допустимых ин- дексов возбуждается исключительная ситуация IndexOutOfBoundsException — *см. разд. 8.8*;

s1.subSequence(index1,index2) — то же, что предыдущий метод, но возвращаемое зна- чение имеет более общий тип CharSequence. Поэтому ссылку на возвращаемую строку можно присваивать как переменным типа String, так и переменным типа StringBuffer и StringBuilder *(см. далее)*, а также подставлять в методы в качестве па- раметров соответствующих типов.

*Важнейшие объектные типы 325*

Поиск подстрок в строке:

s1.indexOf(subS) — индекс позиции, где в строке s1 первый раз встретилась последо-вательность символов subS. Если подстрока не найдена, возвращается значение -1;

s1.indexOf(subS,i) — то же, но поиск начинается с символа, имеющего индекс i;

s1.lastIndexOf(subS) — индекс позиции, где в строке s1 последний раз встретилась последовательность символов subS. Если подстрока не найдена, возвращается значе-ние -1;

s1.lastIndexOf(subS,i) — то же, но поиск вхождения подстроки идет к началу строки с позиции i, а не с конца строки.

Замена подстрок в строке:

s1.replaceFirst(oldSubS,newSubS) — возвращает строку на основе строки s1, в которой произведена замена первого вхождения символов строки oldSubS на символы строки newSubS;

s1.replaceAll(oldSubS,newSubS) — возвращает строку на основе строки s1, в которой произведена замена всех вхождений символов строки oldSubS на символы строки newSubS;

s1.trim() — возвращает копию строки s1, из которой убраны ведущие и завершаю-щие пробелы.

Расщепление строки на массив строк:

s1.split(separator) — возвращает массив строк String[], полученный разделением строки s1 на независимые строки по местам вхождения сепаратора, задаваемого строкой separator. При этом символы, содержащиеся в строке separator, в получив-шиеся строки не входят. Пустые строки из конца получившегося массива удаляют-ся;

s1.split(separator, i) — то же, но положительное i задает максимально допустимое число элементов массива. В этом случае последним элементом массива становится окончание строки s1, которое не было расщеплено на строки вместе с входящими в это окончание символами сепараторов.

При i = 0 ограничений нет, но пустые строки из конца получившегося массива уда-ляются.

При i < 0 ограничений нет, а пустые строки из конца получившегося массива не удаляются.

Работа со строками поддерживается не только классом String, но и многими другими классами. В классе Object имеется метод toString(), обеспечивающий строковое пред-ставление объекта. Конечно, выводимая им строка не может отражать все особенности объекта, а является представлением содержимого объекта по мере возможности. В са-мом классе Object это представление обеспечивает возврат методом полного имени класса (квалифицированное именем пакета), затем идет символ @, после которого сле-дует число — хэш-код объекта (число, однозначно характеризующее данный объект во время сеанса работы) в шестнадцатеричном представлении. Поэтому во всех классах-наследниках, где этот метод не переопределен, он возвращает такую же конструкцию. *326 Глава 8*

Во многих стандартных классах этот метод переопределен. Например, для числовых классов метод toString() обеспечивает вывод строкового представления соответствую-щего числового значения. Для строковых объектов возвращает саму строку, а для сим-вольных (тип char) — символ.

При операциях + и += с операндами, один из которых строковый, а другой нет, метод toString() вызывается автоматически для нестрокового операнда. В результате получа-ется сложение (конкатенация) двух строк.

Для преобразования значения типа char в строку нельзя воспользоваться вызовом мето-да toString(), т. к. соответствующая переменная или литерная константа не является объектом. Но с помощью сложения с пустой строкой можно преобразовать значение типа char в строковое. Пусть ch1 — переменная или значение типа char. Тогда ch1+"" будет объектом типа String.

При сложении числовых величин со строками следует быть очень внимательными, т. к. результат сложения более чем двух слагаемых может оказаться сильно отличающимся от ожидаемого.

Например

String s=1+2+3;

даст вполне ожидаемое значение s=="6".

А вот присваивание

String s="Сумма ="+1+2+3;

даст не очень понятное начинающим программистам значение:

"Сумма =123".

Дело в том, что в первом случае сначала выполняются арифметические сложения, а затем результат преобразуется в строку и присваивается левой части.

А во втором случае сначала производится сложение:

"Сумма ="+1.

Первый операнд строковый, а второй — числовой. Поэтому для второго операнда вы-зывается метод toString() и складываются две строки.

Результатом будет строка:

"Сумма =1".

Затем складывается строка "Сумма =1" и число 2. Опять для второго операнда вызывается метод toString() и складываются две строки.

Результатом будет строка:

"Сумма =12".

Совершенно так же выполняется сложение строки "Сумма =12" и числа 3.

Еще более странный для начинающих программистов результат получится при при-сваивании

String s=1+2+" не равно "+1+2; *Важнейшие объектные типы 327*

Следуя изложенной ранее логике, мы получаем, что результатом будет строка:

"3 не равно 12"

Совместная работа с типами String и char требует особой аккуратности. Например, не-смотря на то, что тип char является числовым, и 'S'==83, в результате присваивания

String s='S'+ "";

в переменной s окажется значение "S", а не "83". Это происходит из-за того, что символ 'S' сначала автоматически преобразуется в объект оболочечного типа Character, кото-рый методом toString() возвращает строковое значение "S".

А в результате присваивания

String s=0+'S'+ "";

в переменной s окажется значение "83", поскольку 0+'S' является числом типа int и имеет значение 83. А затем сумма 83+"" преобразуется в строку "83".

Аналогично, в результате присваивания

String s='S'+ "t"+"ring";

в переменной s окажется значение "String". А в результате присваивания

String s='S'+'t'+ "ring";

в переменной s окажется значение "199ring", т. к. сначала складываются два числовых значения — 'S', равное 83, и 't', равное 116. Затем получившееся число 199 складывает-ся со строкой "ring" и получается "199ring".

До сих пор были приведены простейшие примеры, для которых все достаточно очевид-но. Если же в такого рода выражениях присутствуют числовые и строковые функции, да еще с оператором ? : , то результат может оказаться совершенно непредсказуемым.

Кроме указанных, имеется ряд строковых операторов, заданных в оболочечных число-вых классах. Например, мы уже хорошо знаем методы преобразования строковых пред-ставлений чисел в числовые значения:

Byte.parseByte(строка)

Short.parseShort(строка)

Integer.parseInt(строка)

Long.parseLong(строка)

Float.parseFloat(строка)

Double.parseDouble(строка)

и метод, преобразующий строковые представления чисел в числовые объекты, — экземпляры оболочечных классов Byte, Short, Character, Integer, Long, Float, Double:

valueOf(строка)

Например:

Byte.valueOf(строка)

и т. п.

Кроме того, имеются методы классов Integer и Long для преобразования чисел в двоич-ное и шестнадцатеричное строковое представление: *328 Глава 8*

Integer.toBinaryString(число)

Integer.toHexString(число)

Long.toBinaryString(число)

Long.toHexString(число)

Существует возможность обратного преобразования — из строки в объект соответст-вующего класса (Byte, Short, Integer, Long) с помощью метода decode:

Byte.decode(строка)

и т. п.

Также полезны методы для анализа отдельных символов, имеющиеся в классах String и Character:

s1.charAt(i) — символ в строке s1, имеющий индекс i (индексация начинается с нуля). Метод возбуждает исключительную ситуацию IndexOutOfBoundsException при выходе за пределы допустимых индексов — *см. разд. 8.8*;

s1.indexOf(символ) — индекс позиции, где в строке s1 первый раз встретился символ;

s1.indexOf(символ,i) — индекс позиции, начиная с i, где в строке s1 первый раз встретился символ. Если символ не найден или при выходе за пределы допустимых индексов возвращается значение -1;

s1.lastIndexOf(символ) — индекс позиции, где в строке s1 последний раз встретился символ;

s1.lastIndexOf(символ,i) — то же, но поиск вхождения символа идет к началу строки с позиции i, а не с конца строки;

Character.isDigit(символ) — булева функция, проверяющая, является ли символ циф-рой;

Character.isLetter(символ) — булева функция, проверяющая, является ли символ буквой;

Character.isLetterOrDigit(символ) — булева функция, проверяющая, является ли сим-вол буквой или цифрой;

Character.isLowerCase(символ) — булева функция, проверяющая, набран ли символ в нижнем регистре;

Character.isUpperCase(символ) — булева функция, проверяющая, набран ли символ в верхнем регистре;

Character.isWhitespace(символ) — булева функция, проверяющая, является ли символ "пробелом в широком смысле" (пробелом, символом табуляции, символом перехода на новую строку и т. д.).

Для того чтобы сделать работу с многочисленными присваиваниями более эффектив-ной, используются классы StringBuffer и StringBuilder. Они особенно удобны в тех слу-чаях, когда требуется проводить изменения внутри одной и той же строки (убирать или вставлять символы, менять их местами, заменять одни на другие). Изменение значений переменных этого класса не приводит к созданию мусора, и обычно происходит значи-тельно быстрее, чем при работе с переменными типа String.

В примере работы со строками, который будет рассмотрен в *главе 17*, благодаря ис-пользованию переменных типа StringBuffer вместо переменных типа String удалось по-*Важнейшие объектные типы 329*

высить быстродействие приложения примерно в 30 раз и уменьшить потребление па-мяти в 4000 раз. Эта ситуация типична для задач, требующих многократного изменения содержимого строк.

Класс StringBuffer рекомендуется для тех случаев, когда используются потоки выпол-нения (threads), этот класс, в отличие от String и StringBuilder, обеспечивает синхрони-зацию строк *(см. главу 10)*. Класс StringBuilder, введенный в JDK 5, полностью ему подобен, но синхронизации не поддерживает. Зато обеспечивает бóльшую скорость работы со строками.

К сожалению, совместимости по присваиванию между переменными типа String, StringBuffer и StringBuilder нет. Но в классах StringBuffer и StringBuilder имеется метод sb.append(s), позволяющий добавлять в конец буферизуемой строки sb (имеющей тип StringBuffer или StringBuilder) обычную строку s. Также имеется метод sb.insert(index,s), позволяющий вставлять, начиная с места символа, имеющего индекс index, строку s. Значения типа StringBuffer или StringBuilder преобразуются в тип String с помощью метода toString() (листинг 8.12).

**Листинг 8.12. Работа со строками типа StringBuffer**

StringBuffer sb=new StringBuffer();

sb.append("типа StringBuffer");

sb.insert(0, "Строка ");

System.out.println(sb);

String s=sb.toString();

Кроме строк, в методы append и insert можно подставлять в качестве параметров значе-ния различных примитивных и объектных типов. Значение параметра сначала методом String.valueOf(значение) автоматически преобразуется в значение типа String, после чего выполняется метод append или insert, соответственно, для параметра типа String.

Самый простой способ преобразования объекта-строки в объект типа StringBuffer — это использовать конструктор StringBuffer(String str). Например, в предыдущем фраг-менте кода вместо создания объекта с пустым содержимым и последующего добавле-ния строки методом можно было бы выполнить вызов следующего конструктора:

StringBuffer sb=new StringBuffer("типа StringBuffer");

Обычные строки можно сравнивать с буферизуемыми на совпадение содержания. Для этого предусмотрена булева функция contentEquals.

Вызов s1.contentEquals(sb) возвращает true в случае, когда строка s1 содержит такую же последовательность символов, как и строка sb.

У буферизуемых строк (объектов типа StringBuffer или StringBuilder) помимо длины имеется дополнительное поле capacity ("емкость") — размер буфера под символы стро-ки. Естественно, значение capacity всегда больше или равно длине строки length. При присваиваниях размер буфера при необходимости увеличивается автоматически с не-которым запасом.

Приведем описание важнейших методов для класса StringBuffer (для класса StringBuilder все совершенно аналогично, но, как уже упоминалось, методы не являют-ся синхронизованными). *330 Глава 8*

Конструкторы:

StringBuffer() — создание объекта-строки StringBuffer с пустым содержимым строки и размером буфера в 16 символов.

StringBuffer(capacity) — то же, что в предыдущем случае, но размер буфера задается равным значению параметра capacity, имеющего тип int.

StringBuffer(s1) — создание объекта-строки StringBuffer c копией содержимого объ-екта-строки типа String, на которую ссылается строковая переменная s1.

StringBuffer(sb) — создание объекта-строки StringBuffer c копией содержимого бу-феризуемой строки sb, имеющей тип StringBuffer или StringBuilder.

Конструкторы, позволяющие преобразовывать массивы символов и массывы байтов, а также их подмассивы, в объекты-строки типа StringBuffer, в отличие от класса String, отсутствуют.

Также, в отличие от класса String, в классах StringBuffer и StringBuilder кроме конст-рукторов нет других методов класса.

Функции, аналогичные String.valueOf(параметр) и String.valueOf(charArray,index, count), отсутствуют.

В классах StringBuffer и StringBuilder есть множество методов объектов, хотя пока и не обеспечивающее всех возможностей, имеющихся для класса String. Перечислим важ-нейшие из них, разбив на те же категории, что и для класса String.

Общие характеристики строки:

sb.length() — возвращает длину строки (число 16-битовых символов Unicode, со-держащихся в строке). Длина пустой строки равна нулю;

аналог метода isEmpty(), поддерживаемого классом String, отсутствует, но может быть легко заменен проверкой sb.length()==0;

аналог метода intern(), поддерживаемого классом String, отсутствует.

Преобразование строки в массив или строку другого типа:

sb.getChars(index1,index2,charArray,index) — возвращает массив типа char[], полу-ченный в результате копирования символов строки sb в массив charArray. Копирова-ние начинается с символа, имеющего индекс index1, и заканчивается символом с ин-дексом index2-1. В массиве символы копируются на место, начинающееся с позиции index. Количество скопированных символов равно index2-index1, индекс последнего скопированного символа в массиве charArray равен index+index2-index1-1. При выходе за пределы допустимых индексов возбуждается исключительная ситуация IndexOutOfBoundsException — *см. разд. 8.8*;

sb.toString() — копирование символов строки типа StringBuffer в символы строки типа String;

аналог метода getBytes(charsetName), поддерживаемого классом String, отсутствует.

Преобразование регистра строки:

аналоги методов toLowerCase() и toUpperCase(), поддерживаемых классом String, от-сутствуют.

*Важнейшие объектные типы 331*

Сравнение строк и их частей:

sb.equals(subS) — метод не переопределен, а просто унаследован от класса Object. Поэтому он не только бесполезен, но даже опасен, т. к. вводит программистов в за-блуждение — его лучше заменять эквивалентным оператором сравнения ==. А для сравнения содержимого обычной строки с буферизованной необходимо использо-вать булеву функцию s1.contentEquals(sb1). В классах StringBuffer и StringBuilder метод contentEquals, к сожалению, отсутствует. Поэтому эффективно сравнивать со-держимое этих строк друг с другом пока нельзя;

аналог метода s1.equalsIgnoreCase(s2) отсутствует;

аналоги методов startsWith и endsWith отсутствуют;

аналог метода regionMatches отсутствует.

Копирование части строки:

sb.charAt(index) — возвращает символ с индексом index;

sb.subSequence(index1,index2) — возвращает строку с символами, скопированными из строки sb, начиная с позиции index1 и кончая позицией index2-1. Длина получившей-ся строки равна index2-index1. При выходе за пределы допустимых индексов возбу-ждается исключительная ситуация IndexOutOfBoundsException *(см. разд. 8.8)*. Воз-вращаемое значение имеет тип CharSequence, прародительский для классов String, StringBuffer и StringBuilder. Поэтому ссылку на возвращаемую строку можно при-сваивать как переменным типа String, так и переменным типа StringBuffer или StringBuilder, а также подставлять в методы в качестве параметров соответствующих типов. Метод может быть заменой методу substring *(см. далее)* — для работы без использования строк типа String;

sb.substring(index1) — возвращает строку типа String с символами, скопированны-ми из строки sb, начиная с позиции index1. При выходе за пределы допустимых ин-дексов возбуждается исключительная ситуация StringIndexOutOfBoundsException *(см. разд. 8.8)*;

sb.substring(index1,index2) — возвращает строку типа String с символами, скопиро-ванными из строки sb, начиная с позиции index1 и кончая позицией index2-1. Длина получившейся строки равна index2-index1. При выходе за пределы допустимых ин-дексов возбуждается исключительная ситуация StringIndexOutOfBoundsException *(см. разд. 8.8)*.

Поиск подстрок в строке:

sb.indexOf(subS) — индекс позиции, где в строке sb первый раз встретилась последо-вательность символов подстроки subS, имеющей тип String. Если подстрока не най-дена, возвращается значение -1;

sb.indexOf(subS,i) — то же, но поиск начинается с индекса i;

sb.lastIndexOf(subS) — индекс позиции, где в строке sb последний раз встретилась последовательность символов подстроки subS, имеющей тип String. Если подстрока не найдена, возвращается значение -1;

sb.lastIndexOf(subS,i) — то же, но поиск вхождения подстроки идет к началу строки с позиции i, а не с конца строки.

*332 Глава 8*

Варианты этих методов, использующие поиск по буферизуемой подстроке, отсутст- вуют.

Замена подстрок в строке:

sb.replace(index1,index2,newSubS) — заменяет в строке sb символы в диапазоне от index1 до index2-1 на символы строки newSubS, имеющей тип String. Возвращает ссыл-ку на строку sb (но ее можно не использовать, т. к. она уже хранится в переменной sb). Варианты метода, осуществляющие поиск по буферизуемой подстроке, отсутст-вуют. Данный метод обеспечивает дополнительные возможности по сравнению с методом replaceAll, поддерживаемым классом String;

аналог метода replaceFirst отсутствует;

аналог метода trim() отсутствует.

Расщепление строки на массив строк:

аналог метода split отсутствует.

Несмотря на отсутствие в классах StringBuffer и StringBuilder многих методов, поддер-живаемых классом String, нельзя считать, что эти классы обеспечивают меньшую функциональность, чем String, поскольку в них имеется ряд методов, основанных на изменяемости содержимого строки. С помощью таких изменяемых (mutable) строк можно эффективно выполнять действия, которые невозможно выполнить с помощью неизменяемых (immutable) строк типа String. Перечислим эти методы:

sb.append(v) — добавляет в конец строки строковое представление параметра v, ко-торый может быть любого объектного или примитивного типа. Параметр сначала преобразуется в строку методом String.valueOf(v), после чего результат добавляется в конец строки;

sb.insert(index,v) — вставляет в строку на место с позицией index строковое пред-ставление параметра v, который может быть любого объектного или примитивного типа. Параметр сначала преобразуется в строку методом String.valueOf(v), после чего результат вставляется в соответствующее место строки. При выходе за пределы допустимых индексов возбуждается исключительная ситуация IndexOutOfBoundsException *(см. разд. 8.8)*;

sb.insert(index,sb2,index1,index2) — вставляет в строку sb на место с позицией index часть строки sb2, задаваемую диапазоном индексов от index1 до index2-1. Строка sb2 может иметь тип String, StringBuffer, StringBuilder или любой другой, являющийся наследником CharSequence. При выходе за пределы допустимых индексов возбужда-ется исключительная ситуация IndexOutOfBoundsException *(см. разд. 8.8)*;

sb.deleteCharAt(index) — удаляет из строки sb символ с индексом index. При выходе за пределы допустимых индексов возбуждается исключительная ситуация StringIndexOutOfBoundsException *(см. разд. 8.8)*;

sb.setCharAt(index,символ) — заменяет в строке sb символ с индексом index на новое значение символ. При выходе за пределы допустимых индексов возбуждается исклю-чительная ситуация IndexOutOfBoundsException — *см. разд. 8.8*;

sb.delete(index1,index2) — удаляет из строки sb символы с индексами от index1 до index2-1. При выходе за пределы допустимых индексов возбуждается исключитель-ная ситуация StringIndexOutOfBoundsException — *см. разд. 8.8*;

*Важнейшие объектные типы 333*

sb.setLength() — установка новой длины строки;

sb.capacity() — возвращает текущий размер буфера под символы строки;

sb.ensureCapacity(n) — установка новой емкости строки (размера буфера), задавае-мой переменной n, имеющей тип int. Если текущий размер буфера равен old\_n сим-волов, то новый размер буфера становится равным наибольшему из двух значений: n и 2\*old\_n+2. Если n отрицательно или равно нулю, никаких действий не производит-ся;

sb.trimToSize() — пытается уменьшить размер буфера строки sb до минимально до-пустимого размера;

sb.reverse() — инвертирует последовательность символов в строке sb, т. е. первый символ становится последним, второй — предпоследним и т. д. Возвращает ссылку на строку sb (но ее можно не использовать, т. к. она уже хранится в переменной sb).

Подведем краткие итоги.

Класс String дает два преимущества по сравнению с StringBuffer и StringBuilder. Во-первых, этот класс обеспечивает наибольшее количество методов для работы со стро-ками, поэтому алгоритмы с использованием строк типа String оказываются более про-стыми и прозрачными. Во-вторых, метод intern() позволяет заметно экономить память при хранении строковых данных с большим числом повторяющихся строк (например, имен людей или почтовых адресов). В то же время крайне неэффективная работа со строками типа String при изменении их содержимого в циклах заметно ограничивает возможности этого класса.

Но неизменяемость объектов-строк типа String приводит к крайне неэффективной ра-боте с памятью и весьма медленной обработке таких строк *(см. главу 17)*. Лучшие ре-зультаты во многих случаях дают экземпляры классов StringBuffer и StringBuilder, но они пока не обеспечивают всей необходимой функциональности, и приходится исполь-зовать строки типа String.

**8.5. Типы-перечисления (enum)**

Иногда требуются элементы, которые не являются ни числами, ни строками, но ведут себя как имена элементов и одновременно обладают порядковыми номерами. Напри-мер, названия месяцев или дней недели. В этих случаях используют *перечисления*. Для задания типа какого-либо перечисления следует написать зарезервированное слово enum (сокращение от enumeration — перечисление), после которого — имя задаваемого типа, а затем — в фигурных скобках через запятую элементы перечисления. В качестве эле-ментов можно использовать любые простые идентификаторы (не содержащие квали-фикаторов вида имя1.имя2).

Тип-перечисление обязан быть глобальным — он может задаваться либо на уровне па-кета, либо в каком-либо классе. Но его нельзя задавать внутри какого-либо метода. Элементы перечисления могут иметь любые имена, в том числе совпадающие в разных перечислениях или совпадающие с именами классов или их членов (каждое перечисле-ние имеет свое собственное *пространство имен*). Доступ к элементу перечисления осуществляется с квалификацией именем типа-перечисления:

ИмяТипа.имяЭлемента *334 Глава 8*

У каждого элемента перечисления имеется порядковый номер, соответствующий его положению в наборе (нумерация начинается с нуля). Поэтому первый элемент имеет номер 0, второй элемент — номер 1 и т. д. Имеется функция ordinal(), возвращающая порядковый номер элемента в перечислении. Также имеется функция compareTo(), по-зволяющая сравнивать два элемента перечисления (она возвращает разницу в их поряд-ковых номерах).

Строковое представление значения можно получить с помощью функции name(). Пре-образование из строки в значение типа "перечисление" осуществляется с помощью функции класса valueOf, в которую передается строковое представление значения.

Если требуется рассматривать элементы перечисления как массив, можно воспользо-ваться функцией values() — она возвращает массив элементов, к которым можно об-ращаться по индексу.

Формат вызова функции такой:

ИмяТипа.values()

Для примера зададим типы-перечисления Months (месяцы) и Spring (весна), соответст-вующие различным наборам месяцев:

enum Months {jan, feb, mar, apr, may, jun, jul, aug, sept, oct, nov, dec}

enum Spring {march, apr, may }

Названия месяцев мы намеренно пишем со строчной буквы, чтобы было понятно, что это идентификаторы переменных, а не типы. А имя месяца марта написано по-разному в типах Months и Spring, чтобы показать независимость их пространств имен.

Переменные типа "перечисление" объявляются так же, как и все остальные типы, при этом переменные могут быть как неинициализированы, так и инициализированы:

public Months m1 ,m2=Months.mar, m3;

(при задании в классе общедоступных полей m1, m2 и m3),

Spring spr1=Spring.apr, spr2;

(при задании в методе локальной переменной или задании в классе поля spr1 с пакет-ным уровнем доступа).

После этого возможны следующие операторы:

spr2=spr1;

spr1=Spring.may;

System.out.println("Результат сравнения="+spr2.compareTo(Spring.march));

В переменной spr2 оказывается значение Spring.apr, порядковый номер которого на единицу больше, чем у значения Spring.march, с которым идет сравнение. Поэтому по-сле выполнения этих операторов в консольное окно будет выведен текст:

Результат сравнения=1

Пусть в переменной spr2 хранится значение Spring.may. Порядковый номер значения, хранящегося в переменной, можно получить с помощью вызова spr2.ordinal(). Он воз-вратит число 2, т. к. may — третий элемент перечисления (сдвиг на единицу получается из-за того, что нумерация начинается с нуля). *Важнейшие объектные типы 335*

Строковое представление значения, хранящегося в переменной spr2, можно получить с помощью вызова spr2.name(). Он возвратит строку may — имя типа в возвращаемое зна-чение не входит.

Если переменная типа "перечисление" не инициализирована, в ней хранится значение null.

Поэтому вызов

System.out.println("spr2="+spr2);

осуществленный до присваивания переменной spr2 значения, возвратит строку:

spr2=null

А вот попытки вызовов spr2.ordinal() или spr2.name() приведут к возникновению ошиб-ки (исключительной ситуации) с диагностикой примерно такого вида:

Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException

Получение значения типа Spring по номеру, хранящемуся в переменной i, осуществля-ется так:

spr1=Spring.values()[i];

Преобразование из строки в значение типа Spring будет выглядеть так:

spr1=Spring.valueOf("march");

Необходимо отметить, что перечисления — объектные типы — наследники класса Enum, несмотря на отличный от других классов синтаксис задания. Элементы перечисления являются переменными класса.

**8.6. Работа с датами и временем. Класс *GregorianCalendar***

Мы уже знаем функцию System.currentTimeMillis(), позволяющую узнать текущее время в миллисекундах, прошедшее с момента 0 часов 0 минут 1 января 1970 года.

В старых версиях Java при работе с временем и датами обычно использовались классы java.util.Time и java.util.Date. В настоящее время почти все методы этих классов обо-значены как устаревшие и нерекомендуемые (deprecated), хотя сами классы пока ис-пользовать можно. Тем не менее, без устаревших методов классы Time и Date нефунк-циональны, и их следует заменять классом java.util.GregorianCalendar.

Конструктор GregorianCalendar() создает объект с датой и временем, соответствующими дате и времени для компьютера, на котором создан объект, на момент создания объек-та. Также имеются варианты конструкторов, позволяющие создавать объекты с задан-ной датой и временем. После вызова метода clear(), очищающего все поля, значения констант сбрасываются в значения по умолчанию.

Из объекта типа GregorianCalendar можно осуществлять доступ к методу get(*имяКонстанты*), возвращающему значение, соответствующее дате и времени этого объекта и тому, какая константа задана в качестве параметра. Имеются следующие константы, с по- мощью которых можно получить информацию о дате и времени: *336 Глава 8*

ERA — эра. При задании константы в качестве параметра метода get возвращаемое значение AD==1 (от латинского Anno Domini — наша эра) или BC==0 (от Before Christ — до Рождества Христова). По умолчанию объект имеет значение эры AC.

YEAR — год. По умолчанию объект имеет значение года 1970.

MONTH — месяц. Возвращаемые значения от JANUARY до DECEMBER (т. е. от 0 до 11). По умолчанию объект имеет значение месяца JANUARY==0.

DAY\_OF\_MONTH — день месяца. Возвращаемые значения от 1 до 31. По умолчанию объ-ект имеет значение дня месяца 1.

DAY\_OF\_WEEK — день недели. Возвращаемые значения от SUNDAY до SATURDAY. По умол-чанию объект имеет значение дня недели SUNDAY==1.

WEEK\_OF\_MONTH — неделя месяца. Возвращаемые значения от 0 до 6. По умолчанию объект имеет значение недели месяца 0. Номер недели учитывает возможность су-ществования неполных недель, состоящих менее чем из 7 дней.

DAY\_OF\_WEEK\_IN\_MONTH — номер соответствующего дня недели (воскресенья, понедель-ника, вторника и т. п.) в месяце. Например, второе воскресенье месяца. Может при-нимать не только положительные, но и нулевые или отрицательные значения — в этом случае идет отсчет для предыдущих недель. По умолчанию объект имеет значение номера дня недели 1.

AM\_PM — до полудня или после полудня. Возвращаемые значения AM (от латинского Ante Meridiem — до полудня) и PM (от латинского Post Meridiem — после полудня) типа int. По умолчанию имеет значение AM.

HOUR — номер часа для объекта календаря в 12-часовом варианте исчисления време-ни. Возвращаемые значения от 0 до 11.

HOUR\_OF\_DAY — номер часа для объекта календаря в 24-часовом варианте исчисления времени. Возвращаемые значения от 0 до 23.

MINUTE — номер минуты в соответствующем часе. Возвращаемые значения от 0 до 59.

SECOND — номер секунды в соответствующей минуте. Возвращаемые значения от 0 до 59.

MILLISECOND — номер миллисекунды в соответствующей секунде. Возвращаемые зна-чения от 0 до 999.

Дату и время объекта календаря можно изменить с помощью соответствующих сетте-ров:

set(int year, int dayOfMonth, int date)

set(int year, int month, int dayOfMonth, int hourOfDay, int minute)

set(int year, int month, int dayOfMonth, int hourOfDay, int minute, int second)

setTimeInMillis(long millis).

Метод setFirstDayOfWeek(int value) позволяет задавать, какой день недели считается первым. По умолчанию это воскресенье (GregorianCalendar.SUNDAY), как принято в США. В России это понедельник (GregorianCalendar.MONDAY), поэтому необходимо задать

calendar.setFirstDayOfWeek(GregorianCalendar.MONDAY); *Важнейшие объектные типы 337*

Следует отметить, что эта установка влияет на значение, возвращаемое не методом calendar.get(DAY\_OF\_WEEK), а методом calendar.getFirstDayOfWeek().

Также имеется возможность установки временной зоны и ряда других параметров.

В качестве примера работы с датой и временем, а также перечислениями приведем код класса GUI\_application (листинг 8.13) из проекта 08\_6\_DateTime с прилагаемого к книге DVD.

**Листинг 8.13. Иллюстрация работы с датой, временем и перечислениями**

package javaapplication;

import java.util.GregorianCalendar;

import static java.util.GregorianCalendar.\*;

public class GUI\_application extends javax.swing.JFrame {

enum WeekDays {

Воскресенье,

Понедельник,

Вторник,

Среда,

Четверг,

Пятница,

Суббота;

}

public static void main(String[] args) {

GregorianCalendar calendar = new GregorianCalendar();

//calendar.setTime(new Date());

System.out.println("Сегодня эра "+calendar.get(ERA));

System.out.println("Сегодня год номер "+calendar.get(YEAR));

System.out.println("Сегодня месяц номер "+calendar.get(MONTH));

System.out.println("День недели номер "+

calendar.get(DAY\_OF\_WEEK));

calendar.setFirstDayOfWeek(MONDAY);

System.out.println("Номер дня недели не изменился: " +

calendar.get(DAY\_OF\_WEEK));

System.out.println("Сегодня день месяца номер "+

calendar.get(DAY\_OF\_MONTH));

System.out.println("Сегодня день года номер "+

calendar.get(DAY\_OF\_YEAR));

System.out.println("Час "+calendar.get(HOUR));

System.out.println("Минута "+calendar.get(MINUTE));

System.out.println("В РФ день недели под названием " +

calendar.getDisplayName(DAY\_OF\_WEEK,LONG,

new java.util.Locale("RU")));

System.out.println("В США день недели под названием " +

calendar.getDisplayName(DAY\_OF\_WEEK,LONG,java.util.Locale.US)); *338 Глава 8*

WeekDays w=WeekDays.values()[calendar.get(DAY\_OF\_WEEK)-1];

System.out.println("Сегодня день недели под названием "+w);

String s=w.name();

System.out.println("Сегодня день недели под названием "+s);

}

}

Проанализируем данный фрагмент.

Во-первых, обратим внимание, что вместо того, чтобы использовать вызовы вида calendar.get(GregorianCalendar.ERA), calendar.get(GregorianCalendar.YEAR) и т. п., мы им-портировали все содержащиеся в классе GregorianCalendar константы (статические поля).

Во-вторых, мы применили два различных способа для вывода в консоль названия дня недели:

Метод getDisplayName возвращает строку с названием дня недели в полном (констан-та LONG в качестве параметра) или сокращенном виде (константа SHORT) с учетом за-данного языкового формата (locale). Для России языковый формат приходится соз-давать с помощью конструктора, а для ряда других стран можно задавать его в более простом виде: Locale.US для США, Locale.FR для Франции, Locale.CHINA для Китая и т. д.

Мы задали перечисление с русскими названиями элементов и использовали эти на-звания для вывода. При этом для нахождения нужного нам дня недели пришлось учесть, что дни недели нумеруются от 1 до 7, а элементы нашего перечисления — от 0 до 6. В последних трех строчках листинга 8.11 иллюстрируется как автоматиче-ское, так и явное (с помощью метода name()) преобразование имени перечисления в строку.

Наконец, помимо перечисленных классов для отсчета времени часто используют класс java.util.Timer. Он работает только в многопоточном режиме *(см. главу 10)*.

**8.7. Работа с графикой и графическим пользовательским интерфейсом**

**Код визуализации и код бизнес-логики приложения**

Код любой программы с графическим пользовательским интерфейсом состоит из двух частей — кода, относящегося к взаимодействию с элементами этого интерфейса, и ко-да, связанного с выполнением основных действий приложения. Например, проведение вычислений; управление устройствами, объектами или базами данных; запись в файлы или чтение из файлов и т. д.

Код, относящийся к работе с пользовательским интерфейсом, называется кодом *визуа-лизации* или кодом *представления данных*. А код, отвечающий за выполнение основ-ных действий приложения, называется кодом *бизнес-логики*. Он отвечает за получение и обработку данных.

В правильно написанной программе их следует разделять. Это связано с тем, что зада-чи, связанные с получением и обработкой данных, не имеют никакого отношения к *Важнейшие объектные типы 339*

представлению этих данных в том или ином виде. Вычисление интеграла не имеет никакого отношения к тому, как мы захотим увидеть результаты этого вычисления. Причем очевидно, что код бизнес-логики меньше подвержен изменениям, чем код визуализации.

Как будет выглядеть пользовательский интерфейс, зависит от того, на какого пользова-теля он рассчитан, какие устройства ввода/вывода используются, и даже от художест-венных вкусов дизайнера.

Код бизнес-логики, напротив, должен быть одним и тем же для разных вариантов поль-зовательского интерфейса, поэтому он должен быть отделен от кода визуализации. Код визуализации всегда использует код бизнес-логики, но не наоборот. Обращение к коду визуализации из кода бизнес-логики категорически запрещено.

Несмотря на простоту и очевидность изложенного правила, на практике часто бывает очень трудно ему следовать. Основная рекомендация — для реализации любых сколь-ко-нибудь значительных вычислительных или управляющих алгоритмов следует пи-сать отдельные классы, а не реализовывать их в обработчиках событий или методах приложения. В простейших учебных примерах мы не будем следовать этому правилу, но в более сложных всегда будем им руководствоваться.

**Графические примитивы**

Вывод графики осуществляется с помощью объектов типа java.awt.Graphics. Для них определен ряд методов, описанных табл. 8.2.

Подразумевается, что:

w — ширина области или фигуры;

h — высота;

x, y — координаты левого верхнего угла области (для фигуры это координаты левого верхнего угла прямоугольника, в который вписана фигура).

|  |  |
| --- | --- |
| ***Таблица 8.2.*** *Методы, с помощью которых осуществляется вывод графики* | |
| **Метод** | **Что делает** |
| **Параметры вывода графики** | |
| Color getColor() | Возвращает текущий цвет рисования |
| setColor(Color c) | Устанавливает текущий цвет рисования |
| Font getFont() | Возвращает текущий шрифт для вывода текстовой информации |
| setFont(Font f) | Устанавливает текущий шрифт для вывода текстовой информации.  Экземпляр шрифта создается с помощью конструкто-ра Font(имяШрифта,стильШрифта,размерШрифта) |
| FontMetrics getFontMetrics() | Возвращает параметры текущего шрифта |
| FontMetrics getFontMetrics(Font f) | Возвращает параметры для произвольного шрифта f |