EPAM Systems, RD Dep.

Конспект и раздаточный материал

JAVA.SE.03 Information Handling

REVISION HISTORY					
Ver.	**Description of Change**	**Author**	**Date**	**Approved**	
				Name	**Effective Date**
<1.0>	Первая версия	Игорь Блинов	<06.09.2011>		
<2.0>	Вторая версия. Конспект переделан под обновленное содержание материала модуля.	Ольга Смолякова	<05.03.2014>		

Содержание
JAVA.SE.03 INFORMATION HANDLING

Класс String

Строка – объект класса **String.**

Ссылка типа **String** на строку-константу:

```
String s = "Это строка";
```

Замечание:

- Пустая строка **String s = ""**; не содержит ни одного символа.
- Пустая ссылка **String s = null**; не указывает ни на какую строку и не является объектом.

Строка является **неизменяемой** (**immutable**). Строковое значение не может быть изменено после создания объекта при помощи какого-либо метода.

Любое изменение приводит к созданию **нового** объекта.

Ссылку на объект класса **String** можно изменить так, чтобы она указывала на другой объект, т.е. на другое значение.

Некоторые конструкторы класса String.

```
String()
String(String str)
String(char[] value)
String(char[] value, int offset, int count)
String(StringBuilder builder)
String(StringBuffer buffer)
```

Примеры создания строк

```
package _java._se._03._string;
import java.io.UnsupportedEncodingException;

public class StringExample {
	public static void main(String[] args) throws UnsupportedEncodingException{
		String str1 = new String();
		char[] data1 = { 'a', 'b', 'c', 'd', 'e', 'f' };
		String str2 = new String(data1, 2, 3);
		char[] data2 = { '\u004A', '\u0062', 'V', 'A' };
		String str3 = new String(data2);
		byte ascii[] = { 65, 66, 67, 68, 69, 70 };
		String str4 = new String(ascii); // "ABCDEF"
		byte[] data3 = { (byte) 0xE3, (byte) 0xEE };
		String str5 = new String(data3, "CP1251");// "го"
		String str6 = new String(data3, "CP866");// "ую"

		System.out.println(str5);
		System.out.println(str6);
	}
}
```

Интерфейс CharSequence

Интерфейс **CharSequence** реализуют классы **String**, **StringBuilder**, **StringBuffer**. Методы интерфейса **CharSequence**:

Метод	Описание
public char charAt(int index)	Возвращает char-значение, находящееся в элементе с указанным индексом. Индекс находится в диапазоне от нуля до length() - 1.
public int length()	Возвращает длину данной последовательности символов. Длина - это количество 16-битных char-значений в последовательности.
public CharSequence subSequence(int start, int end)	Возвращает новый объект CharSequence, содержащий подпоследовательность данной последовательности. Подпоследовательность начинается с символа, находящегося под указанным стартовым индексом и заканчивается символом под индексом end - 1.
public String toString()	Возвращает строку, содержащую символы в данной последовательности и в том же порядке. Длина строки будет равна длине последовательности.

Методы чтения символов

Методы чтения символов из строки:

- **char charAt(int index)** – возвращает символ по значению индекса;
- **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** - возвращает символьное представление участка строки;
- **int length()** – возвращает длину строки;
- **boolean isEmpty()** – возращает true, если строки не содержит симоволов, и false – в противном случае;
- **int codePointAt(int index)** – возвращает кодовую точку для позиции в строке, заданной параметром index
- **int codePointBefore(int index)** – возвращает кодовую точку для позиции в строке, предшествующей заданной параметром index;
- **int codePointCount(int beginIndex, int endIndex)** – возвращает количество кодовых точек в порции вызывающей строки, расположенной между символьными порциями beginIndex и endIndex-1;
- **int offsetByCodePoints(int index, int codePointOffset)** – возвращает позицию в вызывающей строке, расположенную на расстоянии codePointOffset кодовых точек после начальной позиции, заданной параметром index;

Кодовые точки

Для языка Java **кодовая точка (code point)** — это код символа из диапазона от 0 до 10FFFF, термин **кодовая единица (code unit)** используется для ссылки на 16-битные символы. Символы, имеющие значения, большие, чем FFFF, называются **дополнительными (supplemental character)**.

Представим, что мы имеем строку:

Привет!

которая, в Unicode, соответствует этим семи кодовым точкам:

U+041F U+0440 U+0438 U+0432 U+0435 U+0442 U+0021

```
package _java._se._03._string;
public class CodePointExample {
	public static void main(String[] args) {
		char ch='现';// Unicode code - 73b0; utf8 - E7 8E B0
		String str = new String("现");

		int b = str.getBytes().length;
		System.out.println("String size = " + str.getBytes().length);

		System.out.println(ch);
		System.out.println(str);
	}
}
```

```
package _java._se._03._string;
public class HieroglyphExample {
	public static void main(String[] args) {
		String str = "现已整合";
		System.out.println("Строка - " + str);
		System.out.println("Длина строки - " + str.length());
		System.out.println("Длина строки в байтах - " + str.getBytes().length);
		for (int i = 0; i < str.codePointCount(0, str.length()); i++) {
			int index = str.offsetByCodePoints(0, i);
			int code = str.codePointAt(index);
			int[] mas = { code };
			System.out.println(i + "-й символ: " + Integer.toHexString(code)
					+ " " + new String(mas, 0, mas.length));
		}
		System.out.println();
		int[] mas2 = { 0x3fdc, 0x4010 };
		String str2 = new String(mas2, 0, mas2.length);
		System.out.println("Строка - " + str2);
		System.out.println("Длина строки - " + str2.length());
		System.out.println("Длина строки в байтах - " + str2.getBytes().length);
	}
}
```

Методы чтения символов

- **byte[] getBytes()** – возвращает строку в виде последовательности байт, используя кодировку по умолчанию;
- **byte[] getBytes(Charset charset)** - возвращает строку в виде последовательности байт, используя указанную в параметре кодировку;
- **void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)** - возвращает массив байт dst из подстроки с srcBegin до srcEnd индекса;
- **byte[] getBytes(String charsetName)** - возвращает строку в виде последовательности байт, используя название кодировки.

```
package _java._se._03._string;
import java.io.UnsupportedEncodingException;
import java.nio.charset.Charset;

public class GetBytesCharSet {
```

```
	public static void main(String[] args) throws UnsupportedEncodingException {
		byte[] data3 = { (byte) 0xE3, (byte) 0xEE };
		String str = "Мама мыла раму1!";
		byte[] strCP866 = str.getBytes(Charset.forName("cp866"));
		byte[] strCP1251 = str.getBytes("cp1251");
		for (byte b : strCP866)
			System.out.print(b + " ");
		System.out.println();
		for (byte b : strCP1251)
			System.out.print(b + " ");
		System.out.println();
		System.out.println(new String(strCP866));
		System.out.println(new String(strCP866, "cp866"));
		System.out.println(new String(strCP1251));
	}
}
```

Методы сравнения строк:

- **boolean equals(Object obj)** - проверяет идентична ли строка указанному объекту;
- **boolean equalsIgnoreCase(String str2)** - если строки одинаковы, игнорируя строчные-прописные буквы, то true
- **int compareTo(String str2)** - лексиграфическое сравнение строк;
- **compareToIgnoreCase(String str)** - лексиграфическое сравнение строк без учета регистра символов;
- **boolean contentEquals(CharSequence cs)** – сравнивает строку с объектом типа CharSequence;
- **boolean contentEquals(StringBuffer sb)** – сравнивает строку с объектом типа StringBuffer;
- **String intern()** - занесение строки в пул литералов.

Пул литералов

Пул литералов – это коллекция ссылок на строковые объекты.

Строки, являясь частью пула литералов, размещены в куче, но ссылки на них находятся в пуле литералов.

```
package _java._se._03._string._other;
public class Other {
	public static String hello = "Hello";
}
```

```
package _java._se._03._string;
class Other {
	static String hello = "Hello";
}

public class StringLiteralPool {
	public static void main(String[] args) {
		String s1 = "Hello";
		String s2 = new StringBuffer("He").append("llo").toString();
		String s3 = s2.intern();
		System.out.println("s1 == s2? " + (s1 == s2));
```

```
                System.out.println("s1 == s3? " + (s1 == s3));
                String hello = "Hello", lo = "lo";
                System.out.print((hello == "Hello") + " ");
                System.out.print((Other.hello == hello) + " ");
                System.out.print((_java._se._03._string._other.Other.hello == hello)
                        + " ");
                System.out.print((hello == ("Hel" + "lo")) + " ");
                System.out.print((hello == ("Hel" + lo)) + " ");
                System.out.println(hello == ("Hel" + lo).intern());
        }
}
```

Работа с символами строки

- **String toUpperCase()** - преобразует строку в верхний регистр;
- **String toUpperCase(Locale locale)** - преобразует строку в верхний регистр, используя указанную локализацию;
- **String toLowerCase()** - преобразует строку в нижний регистр;
- **String toLowerCase(Locale locale)** - преобразует строку в нижний регистр, используя указанную локализацию;
- **char[] toCharArray()** - преобразует строку в новый массив символов.

Объединение строк

- **String concat(String str)** или **+**

```
package _java._se._03._string;
public class ConcatExample {
        public static void main(String[] args) {
                String attention = "Внимание: ";
                String s1 = attention.concat("!!!");
                String s2 = attention + "неизвестный символ";
                System.out.println("s1 = " + s1);
                System.out.println("s2 = " + s2);
                String str1 = "2" + 2 + 2;
                String str2 = 2 + 2 + "2";
                String str3 = "2" + (2 + 2);
                System.out.println("str1=" + str1 + "; str2=" + str2 + "; str3=" + str3);
        }
    }
```

```
package _java._se._03._string;
public class StringConcatAndPlus {
        public static void main(String[] args) {
                String s1 = null;
                try {
                        s1.concat("abc");
                } catch (NullPointerException e) {
                        e.printStackTrace();
                }
                System.out.println(s1);
                String s2 = null;
                System.out.println(s2 + "abc");
                // concat() returns new String object only when the length of
                // argument string is > 0.
                String s3 = "Blue";
                String s4 = "Sky!";
                String s5 = s3.concat(s4);
                System.out.println(s5 == s3);
```

```
            String s6 = "abc";
            String s7 = s6.concat("");
            System.out.println(s6 == s7);
        }
}
```

Поиск символов и подстрок

- **int indexOf(int ch)** - поиск первого вхождения символа в строке;
- **int indexOf(int ch, int fromIndex)** - поиск первого вхождения символа в строке с указанной позиции;
- **int indexOf(String str)** - поиск первого вхождения указанной подстроки;
- **int indexOf(String str, int fromIndex)** - поиск первого вхождения указанной подстроки с указанной позиции;
- **int lastIndexOf(int ch)** - поиск последнего вхождения символа;
- **int lastIndexOf(int ch, int fromIndex)** - поиск последнего вхождения символа с указанной позиции;
- **int lastIndexOf(String str)** - поиск последнего вхождения строки;
- **int lastIndexOf(String str, int fromIndex)** - поиск последнего вхождения строки с указанной позиции;
- **String replace(char oldChar, char newChar)** - замена в строке одного символа на другой;
- **String replace(CharSequence target, CharSequence replacement)** - замена одной подстроки другой;
- **boolean contains(CharSequence cs)** - проверяет, входит ли указанная последовательность символов в строку;
- **static String copyValueOf(char[] data)** - возвращает строку, равную символам data;
- **static String copyValueOf(char[] data, int offset, int count)** - возвращает подстроку, равную части символов data;
- **boolean endsWith(String suffix)** - заканчивается ли String суффиксом suffix;
- **boolean startsWith(String prefix)** - начинается ли String с префикса prefix;
- **boolean startsWith(String prefix, int toffset)** - начинается ли String с префикса prefix учитывая смещение toffset.

Извлечение подстрок

- **String trim()** – отсекает на концах строки пустые символы;
- **String substring(int startIndex)** – возвращает подстроку, с startIndex до конца строки;
- **String substring(int startIndex, int endIndex)** – возвращает подстроку с beginIndex до endIndex;
- **CharSequence subSequence(int beginIndex, int endIndex)** – возвращает подпоследовательность типа CharSequence как подстроку с beginIndex до endIndex.

Приведение значений элементарных типов и объектов к строке

- **String toString()** - возвращает саму строку;

- **static String valueOf(Object obj)** - возвращает результат toString для объекта;
- **static String valueOf(char[] charArray)** - возвращает строку, из символов **charArray**;
- **static String valueOf(char[] data, int offset, int count)** - возвращает подстроку, из части символов data;
- **static String valueOf(boolean b)** -.возвращает строку “true” или “false”, в зависимости от b;
- **static String valueOf(char c)** - возвращает строку из символа c;
- **static String valueOf(int i)** - возвращает строку, полученную из i;
- **static String valueOf(long l)** - возвращает строку, полученную из l;
- **static String valueOf(float f)** - возвращает строку, полученную из f;
- **static String valueOf(double d)** - возвращает строку, полученную из d.

Форматирование строк

- **static String format(String format, Object... args)**
- **static String format(Locale l, String format, Object... args)**

(см. класс Formatter)

Сопоставление с образцом

- **boolean regionMatches(boolean ignoreCase, int toffset, String ther, int ooffset, int len)** - сравнивает часть строки с другой строкой, если ignoreCase=true, то игнорирует строчные-прописные буквы;
- **boolean regionMatches(int toffset, String other, int ooffset, int len)** - сравнивает часть строки с другой строкой, len - сколько символов сравнивать;
- **String replace(char oldChar, char newChar)** - возвращает строку, где все символы oldChar заменены на newChar;
- **String replace(CharSequence target, CharSequence replacement)** - возвращает строку, заменяя элементы target на replacement.
- **boolean matches(String regexStr)** - удовлетворяет ли строка указанному регулярному выражению;
- **String replaceFirst(String regexStr, String replacement)** - заменяет первое вхождение строки, удовлетворяющей регулярному выражению, указанной строкой;
- **String replaceAll(String regexStr, String replacement)** - заменяет все вхождения строк, удовлетворяющих регулярному выражению, указанной строкой;
- **String[] split(String regexStr)** - разбивает строку на части, границами разбиения являются вхождения строк, удовлетворяющих регулярному выражению;
- **String[] split(String regexStr, int limit)** - аналогично предыдущему, но с ограничением применения регулярного выражения к строке значением limit. Если limit>0, то и размер возвращаемого массива строк не будет больше limit. Если limit<=0, то регулярное выражение применяется к строке неограниченное число раз.

```
package _java._se._03._string;
public class StringReplaceFirst {
	public static void main(String[] args) {
		String str = "Her name is Tamara. Tamana is a good girl.";
```

```
                String strreplace = "Sonia";
                String result = str.replaceFirst("Tamana", strreplace);
                System.out.println(result);
        }
}
```

```
package _java._se._03._string;
public class StringEquals {
        public static void main(String[] args) {
                String str1 = "Hello";
                String str2 = new String("Hello");
                if (str1 == str2)
                        System.out.println("Equal");
                else
                        System.out.println("Not Equal");

                str2 = str2.intern();
                if (str1 == str2)
                        System.out.println("Equal");
                else
                        System.out.println("Not Equal");

                if (str1.equals(str2))
                        System.out.println("Equal");
                else
                        System.out.println("Not Equal");
        }
}
```

Классы StringBuilder, StringBuffer

Определение

Классы **StringBuilder** и **StringBuffer** по своему предназначению близки к классу **String. Но**, содержимое и размеры объектов классов **StringBuilder** и **StringBuffer** можно изменять!!!

Основным и единственным отличием **StringBuilder от StringBuffer** является потокобезопасность последнего.

Конструкторы класса **StringBuilder:**

- **StringBuilder(String str)** – создает **StringBuilder**, значение которого устанавливается в передаваемую строку, плюс дополнительные 16 пустых элементов в конце строки.
- **StringBuilder(CharSequence charSeq)** – строит **StringBuilder**, содержащий те же самые символы как указанно в CharSequence, плюс дополнительные 16 пустых элементов, конечных CharSequence.
- **StringBuilder(int length)** – создает пустой **StringBuilder** с указанной начальной вместимостью.
- **StringBuilder()** – создает пустой **StringBuilder** со способностью 16 (16 пустых элементов).

Чтение и изменение символов объекта **StringBuilder:**

- **int length()** – созвращает количество символов в строке.
- **char charAt(int index)** – возвращает символьное значение, расположенное на месте index.

- **void setCharAt(int index, char ch)** – символ, расположенный на месте index заменяется символом ch.
- **CharSequence subSequence(int start, int end)** – возвращает новую подстроку.

Отличие от String

ОТЛИЧИЕ объектов класса **String** от объектов классов **StringBuilder**, **StringBuffer:** для класса **StringBuffer** не переопределены методы **equals()** и **hashCode()**, т.е. сравнить содержимое двух объектов невозможно, к тому же хэш-коды всех объектов этого типа вычисляются так же, как и для класса **Object**.

Добавление символов

Добавление символов в объект класса **StringBuilder**. Добавляет аргумент этому **StringBuilder**. Данные преобразовываются в строку прежде, чем операция добавить будет иметь место.

- `StringBuilder append(Object obj)`
- `StringBuilder append(String str)`
- `StringBuilder append(CharSequence charSeq)`
- `StringBuilder append(CharSequence charSeq, int start, int end)`
- `StringBuilder append(char[] charArray)`
- `StringBuilder append(char[] charArray, int offset, int length)`
- `StringBuilder append(char c)`
- `StringBuilder append(boolean b)`
- `StringBuilder append(int i)`
- `StringBuilder append(long l)`
- `StringBuilder append(float f)`
- `StringBuilder append(double d)`
- `StringBuilder append(StringBuffer sb)`
- `StringBuilder appendCodePoint(int codePoint)`

Вставка символов

Вставка символов в объект **StringBuilder**. Вставляет второй аргумент в **StringBuilder**.. Первый аргумент целого числа указывает индекс, перед которым должны быть вставлены данные. Данные преобразовывают в строку прежде, чем операция вставки будет иметь место.

- `StringBuilder insert(int offset, Object obj)`
- `StringBuilder insert(int dstOffset, CharSequence seq)`
- `StringBuilder insert(int dstOffset, CharSequence seq, int start, int end)`
- `StringBuilder insert(int offset, String str)`
- `StringBuilder insert(int offset, char[] charArray)`
- `StringBuilder insert(int offset, char c)`
- `StringBuilder insert(int offset, boolean b)`
- `StringBuilder insert(int offset, int i)`
- `StringBuilder insert(int offset, long l)`
- `StringBuilder insert(int offset, float f)`
- `StringBuilder insert(int offset, double d)`
- `StringBuilder insert(int index, char[] str, int offset, int len)`

Удаление символов

Удаление символов из объекта StringBuilder.

- **StringBuilder deleteCharAt(int index)** – удаляет символ, расположенный по index.
- **StringBuilder delete(int start, int end)** – удаляет подпоследовательность от start до end-1(включительно) в последовательности символов **StringBuilder's.**
- **StringBuilder reverse()** – полностью изменяет последовательность символов в этом **StringBuilder.**

Управление ёмкостью

- **int capacity()** – возвращает текущую емкость.
- **void ensureCapacity(int minCapacity)** – гарантирует, что вместимость по крайней мере равна указанному минимуму.
- **void trimToSize()** – уменьшает емкость до величины хранимой последовательности.
- **void setLength(int newLength)** – устанавливает длину символьной последовательности. Если newLength - меньше чем length(), последние символы в символьной последовательности являются усеченными. Если newLength больше чем length(), нулевые символы добавляются в конце символьной последовательности.

Другие методы

В классе присутствуют также методы, аналогичные методам класса **String**: **replace**(), **charAt**(), **length**(), **getChars**(), **codePointAt**(int index), **codePointBefore**(int index), **codePointCount**(int beginIndex, int endIndex), **getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin), **indexOf**(String str), **indexOf**(String str, int fromIndex), **lastIndexOf**(String str), **lastIndexOf**(String str, int fromIndex), **offsetByCodePoints**(int index, codePointOffset), **replace**(int start, int end, String str), **substring**(int start), **substring**(int start, int end) , **toString**().

```
package _java._se._03._string;
public class StringBuilderAppend {
	public static void main(String[] args) {
		StringBuilder sb = new StringBuilder();
		sb.append("Java StringBuilder");
		System.out.println("StringBuilder1 : " + sb);
		sb.append(" Example");
		System.out.println("StringBuilder2 : " + sb);
	}
}
```

```
package _java._se._03._string;
public class StringBuilderInsert {
	public static void main(String[] args) {
		StringBuilder sb = new StringBuilder();
		sb.append("Java StringBuilder");
		sb.insert(5, "insert ");
		System.out.println("StringBuilder :" + sb);
	}
}
```

```
package _java._se._03._string;
public class StringBuilderSetcharat {
	public static void main(String[] args) {
		StringBuilder sb = new StringBuilder();
		sb.append("Java tringBuilder");
		sb.setCharAt(5, 'S');
		System.out.println("StringBuilder : " + sb);
	}
```

```
}
```

Форматирование, класс Formatter

Конструкторы

Класс **Formatter** (пакет java.util) - обеспечивает преобразование формата позволяющее выводить числа, строки, время и даты нужном формате.

- **format(String format, Object... args)**
- **format(Locale l, String format, Object... args)**

Конструкторы класса Formatter:

`Formatter()` - автоматически использует локаль по умолчанию и выделяет объект StringBuffer в качестве буфера для хранения отформатированного вывода.

```
Formatter(Locale loc)
Formatter(Appendable target)
Formatter(Appendable target, Locale loc)
Formatter(String filename) throws FileNotFoundException
Formatter(OutputStream outStrm)
Formatter(PrintStream outStrm)
```

Методы, определенные в Formatter

Метод	Описание
void close()	Закрывает вызывающий Formatter. Это вызывает освобождение любых ресурсов, используемых объектом. После закрытия Formatter он не может повторно использоваться.
void flush()	Выталкивает буфер формата. Это заставляет любой вывод, находящийся в данный момент в буфере, записываться по назначению.
Formatter format(String *fmtStr*, Object ... *args)*	Форматирует аргументы, переданные через *args*, *согласно* формальным спецификаторам, содержащимся в *fmtStr*. Возвращает вызывающий объект.
Formatter format(Locale *loc*, String *fmtStr*, Object ... *args)*	Форматирует аргументы, переданные через *args*, согласно формальным спецификаторам, содержащимся в *fmtStr*. Локаль, указанная в *loc*, используется для этого формата. Возвращает вызывающий объект.
IOException ioException()	Если лежащий в основе объект, служащий назначением вывода, генерирует исключение IOException, то это исключение возвращается. В противном случае возвращается null.

Locale locale()	Возвращает локаль вызывающего объекта.
Appendable out()	Возвращает ссылку на лежащий в основе объект — назначение вывода.
String toString()	Возвращает строку, полученную вызовом toString() на целевом объекте. Если это буфер, будет возвращен форматированный вывод.

Для классов **PrintStream** и **PrintWriter** добавлен метод **printf()**. Метод **printf()** автоматически использует класс **Formatter**.

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

Спецификаторы формата

Спецификаторы формата. Общий синтаксис спецификатора формата следующий:

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

Значение аргумента спецификатора формата **conversion** приведены в таблице далее. Кроме строчного написания значения **conversion** *можно использовать* следующие *значения, определяемые прописными буквами*: ‘B’, ‘H’, ‘S’, ‘C’, ‘X’, ‘E’, ‘G’, ‘A’, ‘T’.

Параметр conversion

Спецификатор формата	Выполняемое форматирование
%a	Шестнадцатеричное значение с плавающей точкой
%b	Логическое (булево) значение аргумента
%c	Символьное представление аргумента
%d	Десятичное целое значение аргумента
%h	Хэш-код аргумента
%e	Экспоненциальное представление аргумента
%f	Десятичное значение с плавающей точкой
%g	Выбирает более короткое представление из двух: %e или %f
%o	Восьмеричное целое значение аргумента
%n	Вставка символа новой строки
%s	Строковое представление аргумента
%t	Время и дата
%x	Шестнадцатеричное целое значение аргумента

%%	Вставка знака %

```
package _java._se._03._format;
import java.util.Formatter;
import java.util.Timer;
public class SimpleFormatExample {
	public static void main(String[] args) {
		Formatter formatter = new Formatter();
		boolean b1 = true;
		Boolean b2 = null;
		formatter.format("1. - %b, %b\n", b1, b2);
		System.out.print(formatter);
		System.out.println("---------------------------------");
		Timer t = new Timer();
		formatter.format("2. - %h", t);
		// Integer.toHexString(t.hashCode())
		System.out.println(formatter);
		System.out.println(Integer.toHexString(t.hashCode()));
	}
}
```

Аргумент спецификатора формата [**argument_index**] имеет два вида: **i$** или <.

- **i$** – i (десятичное целое число) - указывает на положение аргумента во множестве параметров переменной длины **format(String format, Object... args)**, начинающемся с положения 1.
- < – указывает на тот же самый аргумент, который использовался в предыдущем спецификаторе формата в форматирующей последовательности, и не может поэтому быть первым в списке спецификаторов формата.

```
package _java._se._03._format;
import java.util.Formatter;

public class ArgumentIndexExample {
	public static void main(String[] args) {
		Formatter formatter = new Formatter();

		double d1 = 16.78967;
		formatter.format("%1$e, %<f, %<g, %<a\n", d1);
		System.out.println(formatter);
	}
}
```

[**flag**] – указывает выравнивание форматируемого аргумента. Значение параметра **flag** приведены в таблице. Комбинация валидных флагов в спецификаторе формата зависит от преобразования.

Flag	Integrals			Floating-point				Description
	d	o	x X	e E	f	g G	a A	
'-'	ok	ok	ok	ok	ok	ok	ok	Выравнивание по левому краю, требует положительного значения width (Также подходит для отображения символов, времени-даты)

‘#’	x	ok	ok	ok	ok	x	ok	Отображает в виде, применяемом для системы счисления и десятичную точку для вещественных
‘+’	ok	x	x	ok	ok	ok	ok	Отображает знак
‘ ‘	ok	x	x	ok	ok	ok	ok	Положительные числа предваряются пробелом
‘0’	ok	ok	ok	ok	ok	ok	ok	Выводит значение, дополненное нулями вместо пробелов, требует положительного значения width
‘,’	ok	x	x	x	ok	ok	x	Числовые значения включают разделители групп, указываемый локалью.
‘(‘	Ok	x	x	ok	ok	ok	x	Отображает отрицательные числа в круглых скобках

- этот флаг может быть применен к **%o, %x, %a, %e** и **%f**. Для **%a, %e** и **%f** флаг **#** гарантирует наличие десятичной точки, даже если нет значащих разрядов после нее. Предварение флагом **#** спецификатора формата **%x**, приведет к тому, что шестнадцатеричное число будет напечатано с префиксом **0x**. Предварение флагом **#** спецификатора **%o** заставит число печататься с ведущими нулями.

```
package _java._se._03._format;
import java.util.Formatter;
public class FlagExample {
	public static void main(String[] args) {
		Formatter formatter = new Formatter();
		int i1 = 17;
		double d1 = 16.78967;
		formatter.format("1. (%%o) %o%n", i1);
		formatter.format("2. (%%a) %a%n", d1);
		formatter.format("3. (%%x) %x%n", i1);
		formatter.format("4. (%%#o) %#o%n", i1);
		formatter.format("5. (%%#a) %#a%n", d1);
		formatter.format("6. (%%#x) %#x%n", i1);
		System.out.println(formatter);
	}
}
```

Width – минимальное число символов, отводимое под представление форматируемого параметра. Спецификатор ширины поля может быть использован со всеми спецификаторами формата, за исключением **%n**.

Precision – имеет формат **.n**, где **n** – число символов в десятичной части числа. Особенности поведения зависят от преобразования. *Спецификатор точности может применяться к спецификаторам формата* **%f, %e, %g** *и* **%s**.

Когда спецификатор точности применяется к данным с плавающей точкой, сформатированным **%f** или **%e**, он определяет количество отображаемых десятичных разрядов после точки. При использовании **%g** спецификатор точности определяет количество значащих разрядов. Примененный к строкам, спецификатор точности устанавливает максимальную длину поля. Например, %5.7s отображает строку не менее

пяти и не более семи символов длиной. Если строка длиннее, чем максимальная ширина поля, последние символы отсекаются.

```
package _java._se._03._format;
import java.util.Formatter;
public class UseFormatterExample {
	public static void main(String[] args) {
		Formatter formatter = new Formatter();
		int i1 = 345;
		double d1 = 16.78967;
		formatter.format("- %-7dok%n", i1);
		formatter.format("- %+7dok%n", i1);
		formatter.format("- % 7dok%n", i1);
		formatter.format("- %07dok%n", i1);
		formatter.format("- %#fok%n", d1);
		formatter.format("- %.2fok%n", d1);
		System.out.println(formatter);
	}
}
```

Форматирование времени и даты

Спецификатор формата	Выполняемое преобразование
%tH	Час (00 - 23)
%tI	Час (1 - 12)
%tM	Минуты как десятичное целое (00 - 59)
%tS	Секунды как десятичное целое (00 - 59)
%tL	Миллисекунды (000 - 999)
%tY	Год в четырехзначное формате
%ty	Год в двузначное формате (00 - 99)
%tB	Полное название месяца (“Январь”)
%tb или %th	Краткое название месяца (“янв”)
%tm	Месяц в двузначном формате (1 - 12)
%tA	Полное название дня недели (“Пятница”)
%ta	Краткое название дня недели (“Пт”)
%td	День в двузначном формате (1 - 31)
%tR	То же что и “%tH:%tM”
%tT	То же что и “%tH:%tM:%tS”
%tr	То же что и “%tI:%tM:%tS %Tp” где %Tp = (AM или PM)

%tD	То же что и “%tm/%td/%ty”
%tF	То же что и “%tY-%tm-%td”
%tc	То же что и “%ta %tb %td %tT %tZ(**EEFT/FET, DST**) %tY”

```
package _java._se._03._format;
import java.util.Calendar;
import java.util.Formatter;
public class DateTimeFormatExample {
	public static void main(String[] args) {
		Formatter formatter = new Formatter();
		Calendar calendar = Calendar.getInstance();
		formatter.format("%tr", calendar);
		System.out.println(formatter);
	}
}
```

Исключения

При работе с классом **Formatter** могут возникнуть следующие исключения. Данные классы исключений являются подклассами класса **IllegalFormatException**.

Format Exception	Meaning
DublicateFormatFlagsException	Flag используется более, чем один раз
FormatFlagsConversionMismatchException	Flag и conversion несовместимы
IllegalFormatConversionException	Тип аргумента несовместим с преобразованием
IllegalFormatFlagsException	Недействительная комбинация флагов
IllegalFormatPresionException	Точность неправильна или недопустима
IllegalFormatWidthException	Значение width недопустимо
MissingFormatArgumentException	Ошибка при передаче параметров
MissingFormatWidthException	Ошибка при задании ширины
UnknownFormatConversionException	Преобразование неизвестно
UnknownFormatFlagsException	Флаг неизвестен

Printf()

Метод **printf()** автоматически использует объект типа **Formatter** для создания форматированной строки. Она выводится как строка в стандартный поток вывода по умолчанию на консоль.

Метод **printf()** определен в классах **PrintStream** и **PrintWriter**.

В классе **PrintStream** у метода **printf()** две синтаксические формы записи:

- **PrintStream printf(String *fmtString,* Object...*args)***
- **PrintStream printf(Local *loc,* String *fmtString,* Object...*args)***

```
package _java._se._03._format;
import java.util.Calendar;
import java.util.Locale;
public class PringfExample {
	public static void main(String[] args) {
		Calendar cal = Calendar.getInstance();
		System.out.printf(Locale.FRANCE, "%1$tB %1$tA%n", cal);
		System.out.printf(Locale.getDefault(), "%1$tB %1$tA%n", cal);
	}
}
```

Интернационализация

Определение

Интернационализация программы (i18n) –

- Написание программы, работающей в различных языковых окружениях.

Локализация программы (l10n) –

- Адаптация интернационализированной программы к конкретным языковым окружениям.

Пакеты

- java.util
- java.text

Класс Locale

Класс **Locale**, (пакет java. util) идентифицирует используемое языковое окружение.

Локаль определяется:

1) константами: **Locale.US, Locale.FRANCE**

2) конструкторами класса **Locale**

- `Locale(language)` – по языку
- `Locale(language, country)` – по языку и стране
- `Locale(language, country, variant)` – по языку стране и варианту

```
Locale l = new Locale("ru","RU");
Locale l = new Locale("en","US","WINDOWS");
```

Пример	
en_UK_windows	**en_UK_unix**
choose the folder containing colour information	choose the directory containing colour information
en_US	**ru_RU_unix**

choose the folder containing color information	Выберите каталог, содержащий цветовую информацию

Методы класса **Locale**

- **getDefault()** – возвращает текущую локаль, сконструированную на основе настроек операционной системы;
- **getLanguage()** – код языка региона;
- **getDisplayLanguage()** – название языка;
- **getCountry()** – код региона;
- **getDisplayCountry()** – название региона;
- **getAvailableLocales()** – список доступных локалей.

```
package _java._se._03._locale;
import java.util.Locale;
public class LocaleExample {
	public static void main(String[] args) {
		Locale defaultLocale = Locale.getDefault();
		Locale rusLocale = new Locale("ru", "RU");
		Locale usLocale = new Locale("en", "US");
		Locale frLocale = new Locale("fr", "FR");

		System.out.println(defaultLocale.getDisplayCountry());
		System.out.println(defaultLocale.getDisplayCountry(Locale.FRENCH));
		System.out.println(frLocale.getDisplayCountry(defaultLocale));

		System.out.println(usLocale.getDisplayName());
		System.out.println(usLocale.getDisplayName(frLocale));

		System.out.println(rusLocale.getDisplayName(frLocale));

		System.out.println(defaultLocale.getCountry());
		System.out.println(defaultLocale.getLanguage());
		System.out.println(defaultLocale.getVariant());
	}
}
```

Числа и даты

Интернационализация чисел и дат - вывод данных в соответствии с языковым контекстом.

Типы данных

- Числа.
- Время и дата.
- Сообщения.

Пакет

- java.text

Класс **NumberFormat**

Получение форматировщиков чисел:

- getNumberInstance(locale) – обычные числа;
- getIntegerInstance(locale) – целые числа (с округлением);
- getPercentInstance(locale) – проценты;
- getCurrencyInstance(locale) – валюта.

Методы форматирования:

- String format(long) – форматировать целое число;
- String format(double) – форматировать число с плавающей точкой;
- Number parse(String) – разобрать локализованное число.

Выбрасываемое исключение

- ParseException – ошибка разбора.

```
package _java._se._03._locale;
import java.text.NumberFormat;
import java.util.Locale;
public class NumberFormatExample {
	public static void main(String[] args) {
		int data[] = { 100, 1000, 10000, 1000000 };
		NumberFormat nf = NumberFormat.getInstance(Locale.US);
		for (int i = 0; i < data.length; ++i) {
			System.out.println(data[i] + "\t" + nf.format(data[i]));
		}
	}
}
```

```
package _java._se._03._locale;
import java.text.NumberFormat;
import java.util.Locale;
public class NumberFormatWithLocale {
	public static void main(String[] args) {
		double number = 9876.598;

		NumberFormat nfGer = NumberFormat.getNumberInstance(Locale.GERMANY);
		NumberFormat nfJap = NumberFormat.getNumberInstance(Locale.JAPANESE);
		NumberFormat nfDef = NumberFormat.getNumberInstance(Locale.FRANCE);
		System.out.println("Formatting the number: " + nfGer.format(number));
		System.out.println("Formatting the number: " + nfJap.format(number));
		System.out.println("Formatting the number: " + nfDef.format(number));
	}
}
```

```
package _java._se._03._locale;
import java.text.NumberFormat;
import java.util.Locale;
public class CurrencyFormatWithLocale {
	public static void main(String[] args) {
		double number = 9876.598;

		NumberFormat cfGer = NumberFormat.getCurrencyInstance(Locale.GERMANY);
		NumberFormat cfNor = NumberFormat.getCurrencyInstance(
                                                    new Locale("no", "NO"));
		NumberFormat cfUS = NumberFormat.getCurrencyInstance(Locale.US);
		System.out.println(": " + cfGer.format(number));
		System.out.println(": " + cfNor.format(number));
		System.out.println(": " + cfUS.format(number));
	}

}
```

```
package _java._se._03._locale;
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;
public class NumAndCurParser {
	public static void main(String[] args) throws ParseException {
		String numGer = "9.876,598";
```

```
                String curGer = "9.876,60 €";
                NumberFormat nfGer = NumberFormat.getNumberInstance(Locale.GERMANY);
                NumberFormat cfGer = NumberFormat.getCurrencyInstance(Locale.GERMANY);
                double dGer = (Double) nfGer.parse(numGer);
                double dcGer = (Double) cfGer.parse(curGer);
                System.out.println(dGer + " " + dcGer);
        }
}
```

Класс **DateFormat**

Получение форматировщиков времени и даты:

- getDateInstance([dateStyle[, locale]]) – даты;
- getTimeIntance([timeStyle[, locale]]) – времени;
- getDateTimeIntance([dateStyle, timeStyle, [locale]]) – даты и времени.

Стили

- DEFAULT, FULL, LONG, MEDIUM, SHORT

Методы форматирования

- String format(date) – форматировать дату/время
- Date parse(String) – разобрать локализованную дату/время

Выбрасываемое исключение

- ParseException – ошибка разбора

```
package _java._se._03._locale;
import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;
public class DateTimeFormatWithLocale {
        public static void main(String[] args){
                Date date = new Date();
                DateFormat dfUSLong = DateFormat.getDateInstance(
                                                        DateFormat.LONG, Locale.US);
                DateFormat dfUSShort = DateFormat.getDateInstance(
                                                        DateFormat.SHORT, Locale.US);

                System.out.println(dfUSLong.format(date));
                System.out.println(dfUSShort.format(date));
        }
}
```

ResourceBundle

Определение

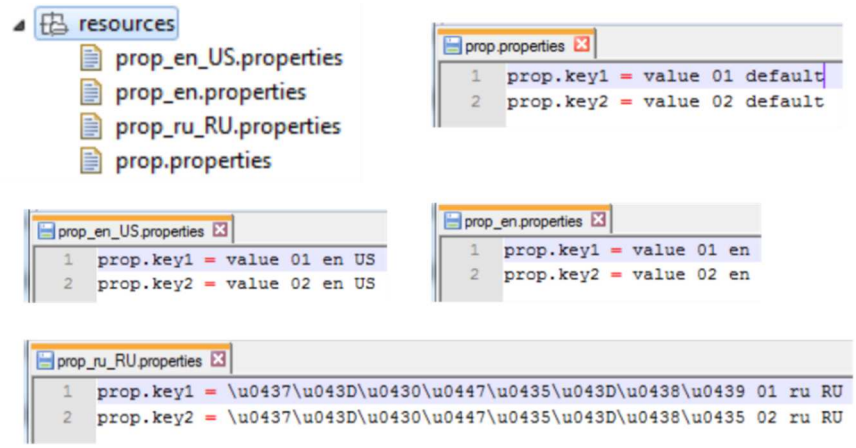
Управление набором ресурсов производится классом **ResourceBundle**, находящимся в пакете **java.util**.

Основой процесса работы с набором ресурсов является получение набора параметров “ключ-значение” при помощи метода **getBundle()** класса **ResourceBundle**.

Ресурс **ResourceExample** может быть представлен либо в виде класса унаследованного от **ListResourceBundle** либо в виде файла, именуемого **ResourceExample.properties**, содержащего пары ключ-значение.

```
package _java._se._03._resourcebundle;
import java.util.ListResourceBundle;
public class AppResources extends ListResourceBundle{
	public Object[][] getContents() {
	    return new Object[][] {
	             { "prop.key1", "value01" },
	             { "prop.key2", "value02" },
	    };
	}
}
```

```
package _java._se._03._resourcebundle;
import java.util.ResourceBundle;
public class UseResourceBundle {
	public static void main(String[] args) {
		ResourceBundle bundle;
		String key = "prop.key1";
		bundle = ResourceBundle.getBundle(
				"_java._se._03._resourcebundle.AppResources");
		System.out.println(bundle.getString(key));
	}
}
```



Для корректного отображения нелатинских символов ознакомьтесь с работой утилиты native2ascii.

```
package _java._se._03._locale;
import java.util.Locale;
import java.util.ResourceBundle;
public class ResourceProperty {
	private ResourceBundle bundle;
	public ResourceProperty(Locale locale) {
		bundle = ResourceBundle
				.getBundle("resources.prop", locale);
	}
	public String getValue(String key) {
		return bundle.getString(key);
	}
}
```

```
package _java._se._03._locale;
import java.util.Locale;
```

```
public class UsePropertiesFromFile {
	public static void main(String[] args) {
		ResourceProperty myBundle = new ResourceProperty(new Locale("en", "US"));
		System.out.println(myBundle.getValue("prop.key1"));
		myBundle = new ResourceProperty(new Locale("en", "UK"));
		System.out.println(myBundle.getValue("prop.key2"));
		myBundle = new ResourceProperty(new Locale("ru", "BY"));
		System.out.println(myBundle.getValue("prop.key1"));
		myBundle = new ResourceProperty(new Locale("ru", "RU"));
		System.out.println(myBundle.getValue("prop.key2"));
	}
}
```

Регулярные выражения

Определение

Регулярные выражения (англ. regular expressions) — современная система поиска текстовых фрагментов в электронных документах, основанная на специальной системе записи образцов для поиска.

В стандартную библиотеку Java входит пакет, специально предназначенный для работы с регулярными выражениями – **java.util.regex**.

Эта библиотека может быть использована для выполнения таких задач:

- поиск данных;
- проверка данных;
- выборочное изменение данных;
- выделение фрагментов данных;
- и др.

Регулярное выражение представляет собой строку-образец (англ. Pattern), состоящую из символов и метасимволов и задающую правило поиска.

Метасимволы:

\	[]
^	-
$	.
?	*
+	()
|	

Символы регулярных выражений.

- **x** – неметасимвол
- \\ - \ как неметасимвол
- **\t** – символ табуляции (‘\u009’)
- **\n** – символ новой строки (‘\u000A’)

- **\r** – символ возврата каретки (‘\u000D’)
- **\f** – символ перевода страницы (‘\u000C’)

Классы символов регулярных выражений.

- **[abc]** – a, b, или c
- **[^abc]** – символ, исключая a, b или c
- **[a-zA-Z]** – символ от a до z или от A до Z, (диапазон)
- **[a-d[m-p]]** – от a до d или от m до p: [a-dm-p] (объединение)
- **[a-z&&[def]]** – d, e, или f (пересечение)
- **[a-z&&[^bc]]** – от a до z, исключая b и c: [ad-z] (вычитание)
- **[a-z&&[^m-p]]** – от a до z, не включая от m до p: [a-lq-z](вычитание)

Предопределенные классы символов.

- **.** – любой символ
- **\d** – цифра [0-9]
- **\D** – не цифра: [^0-9]
- **\s** – [\t\n\x0B\f\r]
- **\S** – [^\s]
- **\w** – [a-zA-Z_0-9]
- **\W** – [^\w]

Обнаружение совпадения вначале и в конце.

- **^a** – якорь для обнаружения сначала строки
- **a$** – якорь на совпадение в конце строки

Логические операторы в регулярных выражениях.

- **ab** – за a следует b
- **a|b** – a либо b
- **(a)** – a, для выделения групп

Квантификаторы.

- ***a*?** – *a* один раз или ни разу
- ***a**** – *a* ноль или более раз
- ***a*+** – *a* один или более раз
- ***a*{*n*}** – *a* *n* раз
- ***a*{*n*,}** – *a* *n или более раз*
- ***a*{*n*,*m*}** – *a* от *n* до *m* раз

Примеры

.+ – будет соответствовать любому тексту
A.+ – любое выражение, которое начинается на букву "A".
^\s+ – один или более пробелов вначале
\s+$ – один или более пробелов вконце

[\d\s()\-]+ – класс символов, в который входят все цифры \\d, все пробельные символы \\s, круглые скобки и дефис. Знак + в конце выражения означает, что любой из этих символов, может встречаться один или более раз.

[a-zA-Z]{1}[a-zA-Z\d\u002E\u005F]+@([a-zA-Z]+\u002E){1,2}((net)|(com)|(org)) - последовательность вида [a-zA-Z] указывает на множество, {n} говорит о том, что некоторый символ должен встретится n раз, а {n,m} - от n до m раз, символ \d указывает на множество цифр, “\u002E” и “\u005F” - это символы точки и подчеркивания соответственно, знак плюс после некоторой последовательности говорит о том, что она должна встретится один или более раз, “|” - представление логического “или”.

([.[^@\s]]+)@([.[^@\s]]+)\.([a-z]+) – формат e-mail адреса

Pattern & Matcher

java.util.regex

Пакет java.util.regex состоит всего из трех классов: **Matcher**, **Pattern**, **PatternSyntaxException**.

- **Pattern** - скомпилированное представление регулярного выражения.
- **Matcher** - движок, который производит операцию сравнения (match).
- **PatternSyntaxException** - указывает на синтаксическую ошибку в выражении.

Последовательность вызова методов при работе с regexp:

```
Pattern p = Pattern.compile("1*0");
Matcher m = p.matcher("111110");
boolean b = m.matches();
```

Методы класса Pattern

- **Pattern compile(String regex)** - возвращает **Pattern,** который соответствует шаблону **regex.**
- **Matcher matcher(CharSequence input)** - возвращает **Matcher,** с помощью которого можно находить соответствия в строке **input.**
- **boolean matches(String regex, CharSequence input)** -проверяет на соответствие строки **input** шаблону **regex.**
- **String pattern()** — возвращает строку, соответствующую шаблону
- **String [ ] split(CharSequence input)** - разбивает строку **input,** учитывая, что разделителем является шаблон.
- **String[] split(CharSequence input, int limit)** -разбивает строку **input** на не более чем **limit** частей.

```
package _java._se._03._pattern;
import java.util.regex.Pattern;
public class PatternExample {
	public static void main(String[] args) {
		String pattern01 = "<+";
		// ревнивая квантификация не только старается
```

```
                // найти максимально длинный вариант,
                String pattern02 = "<?";
                // Использование ленивых квантификаторов может
                // повлечь за собой обратную проблему, когда
                // выражению соответствует слишком короткая, в
                // частности, пустая строка
                String pattern03 = "<*"; // жадный квантияикатор
                String str = "<body><h1> a<<<b </h1></body>";
                String[] result;
                Pattern p = Pattern.compile(pattern01);
                result = p.split(str);
                printTokens(result);
                p = Pattern.compile(pattern02);
                result = p.split(str);
                printTokens(result);
                p = Pattern.compile(pattern03);
                result = p.split(str);
                printTokens(result);
        }
        public static void printTokens(String[] tokens) {
                for (String str : tokens) {
                        if ("".equals(str)) {
                                System.out.print("\"\"" + "|");
                        } else {
                                System.out.print(str + "|");
                        }
                }
                System.out.println();
        }
}
```

Методы класса Matcher

- Начальное состояние объекта типа **Matcher** неопределенно.
- **boolean matches**() — проверяет соответствует ли вся строка шаблону.
- **boolean lookingAt**() — пытается найти последовательность символов, начинающейся с начала строки и соответствующей шаблону.
- **boolean find()** или **boolean find(int start)** - пытается найти последовательность символов соответствующих шаблону в любом месте строки. Параметр **start** указывает на начальную позицию поиска.
- **int end**() — возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону.
- **reset()** или **reset(Char Sequence input)** - сбрасывает состояние **Matcher'a** в исходное, также устанавливает новую последовательность символов для поиска.
- **replaceAll(String replacement)** - замена всех подпоследовательностей символов, удовлетворяющих шаблону, на заданную строку.

Выделение групп

Группы в шаблоне *обозначаются скобками* "(" и ")".

Номера групп начинаются *с единицы*. Нулевая группа совпадает со всей найденной подпоследовательностью.

((A)(B(C)))

- **1** ((A)(B(C)))
- **2** (A)
- **3** (B(C))
- **4** (C)

Методы, для работы с группами

- **String group**() — возвращает всю подпоследовательность, удовлетворяющую шаблону.
- **String group(int group)** — возвращает конкретную группу.
- **int groupCount()** — возвращает количество групп.
- **int end()** — возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону.
- **int end(int group)** — возвращает индекс последнего символа указанной группы.
- **int start()** — возвращает индекс первого символа подпоследовательности, удовлетворяющей шаблону.
- **int start(int group)** — возвращает индекс первого символа указанной группы.

```
package _java._se._03._pattern;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class PatternSplitExample {
	public static void main(String[] args) {
		Pattern p = Pattern.compile("J(\\w*)", Pattern.CASE_INSENSITIVE);
		String text = "Java is fun; JavaScript is funny.; JFunny ; just";
		Matcher m = p.matcher(text);
		while (m.find()) {
			System.out.println("Found '" + m.group(0) + "' at position "
					+ m.start(0) + "-" + m.end(0));
			if (m.start(0) < m.end(0))
				System.out.println("Suffix is " + m.group(1));
		}
	}
}
```

```
package _java._se._03._pattern;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class GroupExample {
	public static void main(String[] args) {
		String text = "test a=\"1\" b=\"2\" c=\"3\" bar d=\"4\" e=\"5\"";
		System.out.println(text + "\n");
		Matcher m1 = Pattern.compile("([a-z]*)(([ \t]+[a-z]=\"[0-9]\")*)")
				.matcher(text);
		/*
		 * Matcher m1 = Pattern .compile(
		 * "([a-z]*)( +[a-z]=\"[0-9]\")( +[a-z]=\"[0-9]\")( +[a-z]=\"[0-9]\")")
		 * .matcher(text);
		 */
		while (m1.find()) {
			System.out.println(m1.group());
			System.out.println(m1.group(1));
			Matcher m2 = Pattern.compile("([a-z])=\"([0-9])\"").matcher(
					m1.group(2));
			while (m2.find()) {
				System.out.println("  " + m2.group(1) + " -> " +
m2.group(2));
			}
```

```
			}
		}
}

package _java._se._03._pattern;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class MatcherLookingAtExample {
	public static void main(String args[]) {
		Pattern p = Pattern.compile("J2SE");

		String candidateString_1 = "J2SE is the only one for me";
		String candidateString_2 = "For me, it's J2SE, or nothing at all";
		String candidateString_3 = "J2SEistheonlyoneforme";

		Matcher matcher = p.matcher(candidateString_1);
		String msg = ":" + candidateString_1 + ": matches?: ";
		System.out.println(msg + matcher.lookingAt());

		matcher.reset(candidateString_2);
		msg = ":" + candidateString_2 + ": matches?: ";
		System.out.println(msg + matcher.lookingAt());

		matcher.reset(candidateString_3);
		msg = ":" + candidateString_3 + ": matches?: ";
		System.out.println(msg + matcher.lookingAt());
	}
}


package _java._se._03._pattern;
import java.util.LinkedList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;
public class Splitter {
	private static final Pattern DEFAULT_PATTERN = Pattern.compile("\\s+");
	private Pattern pattern;
	private boolean keepDelimiters;
	public Splitter(Pattern pattern, boolean keepDelimiters) {
		this.pattern = pattern;
		this.keepDelimiters = keepDelimiters;
	}
	public Splitter(String pattern, boolean keepDelimiters) {
		this(Pattern.compile(pattern == null ? "" : pattern), keepDelimiters);
	}
	public Splitter(Pattern pattern) {
		this(pattern, true);
	}
	public Splitter(String pattern) {
		this(pattern, true);
	}
	public Splitter(boolean keepDelimiters) {
		this(DEFAULT_PATTERN, keepDelimiters);
	}
	public Splitter() {
		this(DEFAULT_PATTERN);
	}
	public String[] split(String text) {
		if (text == null) {
			text = "";
		}
		int last_match = 0;
```

```
                LinkedList<String> splitted = new LinkedList<String>();
                Matcher m = this.pattern.matcher(text);
                while (m.find()) {
                        splitted.add(text.substring(last_match, m.start()));
                        if (this.keepDelimiters) {
                                splitted.add(m.group());
                        }
                        last_match = m.end();
                }
                splitted.add(text.substring(last_match));
                return splitted.toArray(new String[splitted.size()]);
        }
        public static void main(String[] argv) {
                Pattern pattern = null;
                try {
                        pattern = Pattern.compile("\\W+");
                } catch (PatternSyntaxException e) {
                        System.err.println(e);
                        return;
                }
                Splitter splitter = new Splitter(pattern);
                String text = "Hello World!";
                int counter = 1;
                for (String part : splitter.split(text)) {
                        System.out.printf("Part %d: \"%s\"\n", counter++, part);
                }
        }
}
```

Кодировки

Unicode (**Юникод**) — стандарт кодирования символов, позволяющий представить знаки практически всех письменных языков.

Юникод имеет несколько форм представления:

- **UTF-8**;
- **UTF-16** (UTF-16BE(big-endian), UTF-16LE(little-endian)) и
- **UTF-32** (UTF-32BE, UTF-32LE).

Коды в стандарте **Unicode** разделены на несколько областей.

Область с кодами от **U+0000 до U+007F** содержит символы набора **ASCII** с соответствующими кодами.

Далее расположены области знаков различных письменностей, знаки пунктуации и технические символы.

Часть кодов зарезервирована для использования в будущем.

Под символы **кириллицы** выделены коды от **U+0400 до U+052F**.

http://unicode-table.com/ru/#cjk-unified-ideographs

UTF-8 — текст, состоящий только из символов с номером меньше 128, при записи в UTF-8 превращается в обычный текст **ASCII**.

И наоборот, в тексте UTF-8 любой байт со значением меньше 128 изображает символ ASCII с тем же кодом.

Остальные символы Юникода изображаются последовательностями длиной от 2 до 6 байт, в которых первый байт всегда имеет вид **11xxxxxx**, а остальные — **10xxxxxx** (на деле,

только до 4 байт, поскольку в Юникоде нет символов с кодом больше 10FFFF, и вводить их в будущем не планируется).

В потоке данных **UTF-16** старший байт может записываться либо перед младшим (*UTF-16 big-endian*), либо после младшего (*UTF-16 little-endian*). Аналогично существует два варианта четырёхбайтной кодировки — UTF-32BE и UTF-32LE.

Для определения формата представления Юникода в текстовом файле используется приём, по которому в начале текста записывается символ **U+FEFF** (неразрывный пробел с нулевой шириной), также именуемый *меткой порядка байтов* (*byte order mark, BOM*).

Этот способ позволяет различать UTF-16LE и UTF-16BE, поскольку символа U+FFFE не существует. Также он иногда применяется для обозначения формата UTF-8, хотя к этому формату и неприменимо понятие порядка байтов.

Файлы, следующие этому соглашению, начинаются с таких последовательностей байтов:

- **UTF-8** — EF BB BF
- **UTF-16BE** — FE FF
- **UTF-16LE** — FF FE
- **UTF-32BE** — 00 00 FE FF
- **UTF-32LE** — FF FE 00 00

Файлы в кодировках UTF-16 и UTF-32, не содержащие BOM, должны иметь порядок байтов big-endian.

```
package _java._se._03._coding;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
public class UnicodeJava {
	public static void main(String[] args) {
		try {
			InputStreamReader isrBE = new InputStreamReader(
					new FileInputStream("test16be.txt"), "utf16");
			InputStreamReader isrLE = new InputStreamReader(
					new FileInputStream("test16le.txt"), "utf16");
			char[] cbuf = new char[40];
			isrBE.read(cbuf);
			System.out.println(new String(cbuf).trim());
			cbuf = new char[40];
			isrLE.read(cbuf);
			System.out.println(new String(cbuf).trim());
			isrBE.close();
			isrLE.close();
			FileInputStream fisBE = new FileInputStream("test16be.txt");
			FileInputStream fisLE = new FileInputStream("test16le.txt");
			int b;
			while ((b = fisBE.read()) != -1) {
				System.out.print(Integer.toHexString(b) + " ");
			}
			System.out.println();
			while ((b = fisLE.read()) != -1) {
				System.out.print(Integer.toHexString(b) + " ");
			}
			fisBE.close();
			fisLE.close();
		} catch (FileNotFoundException e) {
			e.printStackTrace();
```

```
                } catch (IOException e) {
                      e.printStackTrace();
                }
          }
    }
```