/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Compilation: javac StdIn.java

\* Execution: java StdIn (interactive test of basic functionality)

\* Dependencies: none

\*

\* Reads in data of various types from standard input.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.ArrayList;

import java.util.InputMismatchException;

import java.util.Locale;

import java.util.NoSuchElementException;

import java.util.Scanner;

import java.util.regex.Pattern;

/\*\*

\* The {@code StdIn} class provides static methods for reading strings

\* and numbers from standard input.

\* These functions fall into one of four categories:

\* <ul>

\* <li>those for reading individual tokens from standard input, one at a time,

\* and converting each to a number, string, or boolean

\* <li>those for reading characters from standard input, one at a time

\* <li>those for reading lines from standard input, one at a time

\* <li>those for reading a sequence of values of the same type from standard input,

\* and returning the values in an array

\* </ul>

\* <p>

\* Generally, it is best not to mix functions from the different

\* categories in the same program.

\* <p>

\* <b>Reading tokens from standard input one at a time,

\* and converting to numbers and strings.</b>

\* You can use the following methods to read numbers, strings, and booleans

\* from standard input:

\* <ul>

\* <li> {@link #readInt()}

\* <li> {@link #readDouble()}

\* <li> {@link #readString()}

\* <li> {@link #readBoolean()}

\* <li> {@link #readShort()}

\* <li> {@link #readLong()}

\* <li> {@link #readFloat()}

\* <li> {@link #readByte()}

\* </ul>

\* <p>

\* Each method skips over any input that is whitespace. Then, it reads

\* the next token and attempts to convert it into a value of the specified

\* type. If it succeeds, it returns that value; otherwise, it

\* throws a {@link InputMismatchException}.

\* <p>

\* <em>Whitespace</em> includes spaces, tabs, and newlines; the full definition

\* is inherited from {@link Character#isWhitespace(char)}.

\* A <em>token</em> is a maximal sequence of non-whitespace characters.

\* The precise rules for describing which tokens can be converted to

\* integers and floating-point numbers are inherited from

\* <a href = "http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#number-syntax">Scanner</a>,

\* using the locale {@link Locale#US}; the rules

\* for floating-point numbers are slightly different

\* from those in {@link Double#valueOf(String)},

\* but unlikely to be of concern to most programmers.

\* <p>

\* <b>Reading characters from standard input, one at a time.</b>

\* You can use the following two methods to read characters from standard input:

\* <ul>

\* <li> {@link #hasNextChar()}

\* <li> {@link #readChar()}

\* </ul>

\* <p>

\* The first method returns true if standard input has more input (including whitespace).

\* The second method reads and returns the next character of input on standard

\* input (possibly a whitespace character).

\* <p>

\* As an example, the following code fragment reads characters from standard input,

\* one character at a time, and prints it to standard output.

\* <pre>

\* while (StdIn.hasNextChar()) {

\* char c = StdIn.readChar();

\* StdOut.print(c);

\* }

\* </pre>

\* <p>

\* <b>Reading lines from standard input, one at a time.</b>

\* You can use the following two methods to read lines from standard input:

\* <ul>

\* <li> {@link #hasNextLine()}

\* <li> {@link #readLine()}

\* </ul>

\* <p>

\* The first method returns true if standard input has more input (including whitespace).

\* The second method reads and returns the remaining portion of

\* the next line of input on standard input (possibly whitespace),

\* discarding the trailing line separator.

\* <p>

\* A <em>line separator</em> is defined to be one of the following strings:

\* {@code \n} (Linux), {@code \r} (old Macintosh),

\* {@code \r\n} (Windows),

\* {@code &#92;u2028}, {@code &#92;u2029}, or {@code &#92;u0085}.

\* <p>

\* As an example, the following code fragment reads text from standard input,

\* one line at a time, and prints it to standard output.

\* <pre>

\* while (StdIn.hasNextLine()) {

\* String line = StdIn.readLine();

\* StdOut.println(line);

\* }

\* </pre>

\* <p>

\* <b>Reading a sequence of values of the same type from standard input.</b>

\* You can use the following methods to read a sequence numbers, strings,

\* or booleans (all of the same type) from standard input:

\* <ul>

\* <li> {@link #readAllDoubles()}

\* <li> {@link #readAllInts()}

\* <li> {@link #readAllLongs()}

\* <li> {@link #readAllStrings()}

\* <li> {@link #readAllLines()}

\* <li> {@link #readAll()}

\* </ul>

\* <p>

\* The first three methods read of all of remaining token on standard input

\* and dconverts the tokens to values of

\* the specified type, as in the corresponding

\* {@code readDouble}, {@code readInt}, and {@code readString()} methods.

\* The {@code readAllLines()} method reads all remaining lines on standard

\* input and returns them as an array of strings.

\* The {@code readAll()} method reads all remaining input on standard

\* input and returns it as a string.

\* <p>

\* As an example, the following code fragment reads all of the remaining

\* tokens from standard input and returns them as an array of strings.

\* <pre>

\* String[] words = StdIn.readAllStrings();

\* </pre>

\* <p>

\* <b>Differences with Scanner.</b>

\* {@code StdIn} and {@link Scanner} are both designed to parse

\* tokens and convert them to primitive types and strings.

\* Some of the main differences are summarized below:

\* <ul>

\* <li> {@code StdIn} is a set of static methods and reads

\* reads input from only standard input. It is suitable for use before

\* a programmer knows about objects.

\* See {@link In} for an object-oriented version that handles

\* input from files, URLs,

\* and sockets.

\* <li> {@code StdIn} uses whitespace as the delimiter between tokens.

\* <li> {@code StdIn} coerces the character-set encoding to UTF-8,

\* which is a standard character encoding for Unicode.

\* <li> {@code StdIn} coerces the locale to {@link Locale#US},

\* for consistency with {@link StdOut}, {@link Double#parseDouble(String)},

\* and floating-point literals.

\* <li> {@code StdIn} has convenient methods for reading a single

\* character; reading in sequences of integers, doubles, or strings;

\* and reading in all of the remaining input.

\* </ul>

\* <p>

\* Historical note: {@code StdIn} preceded {@code Scanner}; when

\* {@code Scanner} was introduced, this class was reimplemented to use {@code Scanner}.

\* <p>

\* <b>Using standard input.</b>

\* Standard input is fundamental operating system abstraction, on Mac OS X,

\* Windows, and Linux.

\* The methods in {@code StdIn} are <em>blocking</em>, which means that they

\* will wait until you enter input on standard input.

\* If your program has a loop that repeats until standard input is empty,

\* you must signal that the input is finished.

\* To do so, depending on your operating system and IDE,

\* use either {@code <Ctrl-d>} or {@code <Ctrl-z>}, on its own line.

\* If you are redirecting standard input from a file, you will not need

\* to do anything to signal that the input is finished.

\* <p>

\* <b>Known bugs.</b>

\* Java's UTF-8 encoding does not recognize the optional

\* <a href = "http://bugs.sun.com/bugdatabase/view\_bug.do?bug\_id=4508058">byte-order mask</a>.

\* If the input begins with the optional byte-order mask, {@code StdIn}

\* will have an extra character {@code &#92;uFEFF} at the beginning.

\* <p>

\* <b>Reference.</b>

\* For additional documentation,

\* see <a href="http://introcs.cs.princeton.edu/15inout">Section 1.5</a> of

\* <em>Computer Science: An Interdisciplinary Approach</em>

\* by Robert Sedgewick and Kevin Wayne.

\*

\* @author David Pritchard

\* @author Robert Sedgewick

\* @author Kevin Wayne

\*/

public final class StdIn {

/\*\*\* begin: section (1 of 2) of code duplicated from In to StdIn. \*/

// assume Unicode UTF-8 encoding

private static final String CHARSET\_NAME = "UTF-8";

// assume language = English, country = US for consistency with System.out.

private static final Locale LOCALE = Locale.US;

// the default token separator; we maintain the invariant that this value

// is held by the scanner's delimiter between calls

private static final Pattern WHITESPACE\_PATTERN = Pattern.compile("\\p{javaWhitespace}+");

// makes whitespace significant

private static final Pattern EMPTY\_PATTERN = Pattern.compile("");

// used to read the entire input

private static final Pattern EVERYTHING\_PATTERN = Pattern.compile("\\A");

/\*\*\* end: section (1 of 2) of code duplicated from In to StdIn. \*/

private static Scanner scanner;

// it doesn't make sense to instantiate this class

private StdIn() { }

//// begin: section (2 of 2) of code duplicated from In to StdIn,

//// with all methods changed from "public" to "public static"

/\*\*

\* Returns true if standard input is empty (except possibly for whitespace).

\* Use this method to know whether the next call to {@link #readString()},

\* {@link #readDouble()}, etc will succeed.

\*

\* @return {@code true} if standard input is empty (except possibly

\* for whitespace); {@code false} otherwise

\*/

public static boolean isEmpty() {

return !scanner.hasNext();

}

/\*\*

\* Returns true if standard input has a next line.

\* Use this method to know whether the

\* next call to {@link #readLine()} will succeed.

\* This method is functionally equivalent to {@link #hasNextChar()}.

\*

\* @return {@code true} if standard input has more input (including whitespace);

\* {@code false} otherwise

\*/

public static boolean hasNextLine() {

return scanner.hasNextLine();

}

/\*\*

\* Returns true if standard input has more inputy (including whitespace).

\* Use this method to know whether the next call to {@link #readChar()} will succeed.

\* This method is functionally equivalent to {@link #hasNextLine()}.

\*

\* @return {@code true} if standard input has more input (including whitespace);

\* {@code false} otherwise

\*/

public static boolean hasNextChar() {

scanner.useDelimiter(EMPTY\_PATTERN);

boolean result = scanner.hasNext();

scanner.useDelimiter(WHITESPACE\_PATTERN);

return result;

}

/\*\*

\* Reads and returns the next line, excluding the line separator if present.

\*

\* @return the next line, excluding the line separator if present;

\* {@code null} if no such line

\*/

public static String readLine() {

String line;

try {

line = scanner.nextLine();

}

catch (NoSuchElementException e) {

line = null;

}

return line;

}

/\*\*

\* Reads and returns the next character.

\*

\* @return the next character

\* @throws NoSuchElementException if standard input is empty

\*/

public static char readChar() {

scanner.useDelimiter(EMPTY\_PATTERN);

String ch = scanner.next();

assert ch.length() == 1 : "Internal (Std)In.readChar() error!"

+ " Please contact the authors.";

scanner.useDelimiter(WHITESPACE\_PATTERN);

return ch.charAt(0);

}

/\*\*

\* Reads and returns the remainder of the input, as a string.

\*

\* @return the remainder of the input, as a string

\* @throws NoSuchElementException if standard input is empty

\*/

public static String readAll() {

if (!scanner.hasNextLine())

return "";

String result = scanner.useDelimiter(EVERYTHING\_PATTERN).next();

// not that important to reset delimeter, since now scanner is empty

scanner.useDelimiter(WHITESPACE\_PATTERN); // but let's do it anyway

return result;

}

/\*\*

\* Reads the next token and returns the {@code String}.

\*

\* @return the next {@code String}

\* @throws NoSuchElementException if standard input is empty

\*/

public static String readString() {

return scanner.next();

}

/\*\*

\* Reads the next token from standard input, parses it as an integer, and returns the integer.

\*

\* @return the next integer on standard input

\* @throws NoSuchElementException if standard input is empty

\* @throws InputMismatchException if the next token cannot be parsed as an {@code int}

\*/

public static int readInt() {

return scanner.nextInt();

}

/\*\*

\* Reads the next token from standard input, parses it as a double, and returns the double.

\*

\* @return the next double on standard input

\* @throws NoSuchElementException if standard input is empty

\* @throws InputMismatchException if the next token cannot be parsed as a {@code double}

\*/

public static double readDouble() {

return scanner.nextDouble();

}

/\*\*

\* Reads the next token from standard input, parses it as a float, and returns the float.

\*

\* @return the next float on standard input

\* @throws NoSuchElementException if standard input is empty

\* @throws InputMismatchException if the next token cannot be parsed as a {@code float}

\*/

public static float readFloat() {

return scanner.nextFloat();

}

/\*\*

\* Reads the next token from standard input, parses it as a long integer, and returns the long integer.

\*

\* @return the next long integer on standard input

\* @throws NoSuchElementException if standard input is empty

\* @throws InputMismatchException if the next token cannot be parsed as a {@code long}

\*/

public static long readLong() {

return scanner.nextLong();

}

/\*\*

\* Reads the next token from standard input, parses it as a short integer, and returns the short integer.

\*

\* @return the next short integer on standard input

\* @throws NoSuchElementException if standard input is empty

\* @throws InputMismatchException if the next token cannot be parsed as a {@code short}

\*/

public static short readShort() {

return scanner.nextShort();

}

/\*\*

\* Reads the next token from standard input, parses it as a byte, and returns the byte.

\*

\* @return the next byte on standard input

\* @throws NoSuchElementException if standard input is empty

\* @throws InputMismatchException if the next token cannot be parsed as a {@code byte}

\*/

public static byte readByte() {

return scanner.nextByte();

}

/\*\*

\* Reads the next token from standard input, parses it as a boolean,

\* and returns the boolean.

\*

\* @return the next boolean on standard input

\* @throws NoSuchElementException if standard input is empty

\* @throws InputMismatchException if the next token cannot be parsed as a {@code boolean}:

\* {@code true} or {@code 1} for true, and {@code false} or {@code 0} for false,

\* ignoring case

\*/

public static boolean readBoolean() {

String s = readString();

if ("true".equalsIgnoreCase(s)) return true;

if ("false".equalsIgnoreCase(s)) return false;

if ("1".equals(s)) return true;

if ("0".equals(s)) return false;

throw new InputMismatchException();

}

/\*\*

\* Reads all remaining tokens from standard input and returns them as an array of strings.

\*

\* @return all remaining tokens on standard input, as an array of strings

\*/

public static String[] readAllStrings() {

// we could use readAll.trim().split(), but that's not consistent

// because trim() uses characters 0x00..0x20 as whitespace

String[] tokens = WHITESPACE\_PATTERN.split(readAll());

if (tokens.length == 0 || tokens[0].length() > 0)

return tokens;

// don't include first token if it is leading whitespace

String[] decapitokens = new String[tokens.length-1];

for (int i = 0; i < tokens.length - 1; i++)

decapitokens[i] = tokens[i+1];

return decapitokens;

}

/\*\*

\* Reads all remaining lines from standard input and returns them as an array of strings.

\* @return all remaining lines on standard input, as an array of strings

\*/

public static String[] readAllLines() {

ArrayList<String> lines = new ArrayList<String>();

while (hasNextLine()) {

lines.add(readLine());

}

return lines.toArray(new String[lines.size()]);

}

/\*\*

\* Reads all remaining tokens from standard input, parses them as integers, and returns

\* them as an array of integers.

\* @return all remaining integers on standard input, as an array

\* @throws InputMismatchException if any token cannot be parsed as an {@code int}

\*/

public static int[] readAllInts() {

String[] fields = readAllStrings();

int[] vals = new int[fields.length];

for (int i = 0; i < fields.length; i++)

vals[i] = Integer.parseInt(fields[i]);

return vals;

}

/\*\*

\* Reads all remaining tokens from standard input, parses them as longs, and returns

\* them as an array of longs.

\* @return all remaining longs on standard input, as an array

\* @throws InputMismatchException if any token cannot be parsed as a {@code long}

\*/

public static long[] readAllLongs() {

String[] fields = readAllStrings();

long[] vals = new long[fields.length];

for (int i = 0; i < fields.length; i++)

vals[i] = Long.parseLong(fields[i]);

return vals;

}

/\*\*

\* Reads all remaining tokens from standard input, parses them as doubles, and returns

\* them as an array of doubles.

\* @return all remaining doubles on standard input, as an array

\* @throws InputMismatchException if any token cannot be parsed as a {@code double}

\*/

public static double[] readAllDoubles() {

String[] fields = readAllStrings();

double[] vals = new double[fields.length];

for (int i = 0; i < fields.length; i++)

vals[i] = Double.parseDouble(fields[i]);

return vals;

}

//// end: section (2 of 2) of code duplicated from In to StdIn

// do this once when StdIn is initialized

static {

resync();

}

/\*\*

\* If StdIn changes, use this to reinitialize the scanner.

\*/

private static void resync() {

setScanner(new Scanner(new java.io.BufferedInputStream(System.in), CHARSET\_NAME));

}

private static void setScanner(Scanner scanner) {

StdIn.scanner = scanner;

StdIn.scanner.useLocale(LOCALE);

}

/\*\*

\* Reads all remaining tokens, parses them as integers, and returns

\* them as an array of integers.

\* @return all remaining integers, as an array

\* @throws InputMismatchException if any token cannot be parsed as an {@code int}

\* @deprecated Replaced by {@link #readAllInts()}.

\*/

@Deprecated

public static int[] readInts() {

return readAllInts();

}

/\*\*

\* Reads all remaining tokens, parses them as doubles, and returns

\* them as an array of doubles.

\* @return all remaining doubles, as an array

\* @throws InputMismatchException if any token cannot be parsed as a {@code double}

\* @deprecated Replaced by {@link #readAllDoubles()}.

\*/

@Deprecated

public static double[] readDoubles() {

return readAllDoubles();

}

/\*\*

\* Reads all remaining tokens and returns them as an array of strings.

\* @return all remaining tokens, as an array of strings

\* @deprecated Replaced by {@link #readAllStrings()}.

\*/

@Deprecated

public static String[] readStrings() {

return readAllStrings();

}

/\*\*

\* Interactive test of basic functionality.

\*

\* @param args the command-line arguments

\*/

public static void main(String[] args) {

StdOut.print("Type a string: ");

String s = StdIn.readString();

StdOut.println("Your string was: " + s);

StdOut.println();

StdOut.print("Type an int: ");

int a = StdIn.readInt();

StdOut.println("Your int was: " + a);

StdOut.println();

StdOut.print("Type a boolean: ");

boolean b = StdIn.readBoolean();

StdOut.println("Your boolean was: " + b);

StdOut.println();

StdOut.print("Type a double: ");

double c = StdIn.readDouble();

StdOut.println("Your double was: " + c);

StdOut.println();

}

}