java.io

Class InputStream

java.lang.Object
  └ **java.io.InputStream**

All Implemented Interfaces:

Closeable

Direct Known Subclasses:

AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, InputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream

---

```
public abstract class InputStream
extends Object
implements Closeable
```

This abstract class is the superclass of all classes representing an input stream of bytes.

Applications that need to define a subclass of `InputStream` must always provide a method that returns the next byte of input.

Since:

JDK1.0

See Also:

BufferedInputStream, ByteArrayInputStream, DataInputStream, FilterInputStream, read(), OutputStream, PushbackInputStream

---

Constructor Summary

InputStream()

Method Summary

int	**available**() Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.

void	**close**() Closes this input stream and releases any system resources associated with the stream.
void	**mark**(int readlimit) Marks the current position in this input stream.
boolean	**markSupported**() Tests if this input stream supports the mark and reset methods.
abstract int	**read**() Reads the next byte of data from the input stream.
int	**read**(byte[] b) Reads some number of bytes from the input stream and stores them into the buffer array b.
int	**read**(byte[] b, int off, int len) Reads up to len bytes of data from the input stream into an array of bytes.
void	**reset**() Repositions this stream to the position at the time the mark method was last called on this input stream.
long	**skip**(long n) Skips over and discards n bytes of data from this input stream.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

InputStream

```
public InputStream()
```

Method Detail

read

```
public abstract int read()
                  throws IOException
```

Reads the next byte of data from the input stream. The value byte is returned as an int in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.

A subclass must provide an implementation of this method.

Returns:

the next byte of data, or `-1` if the end of the stream is reached.

Throws:

`IOException` - if an I/O error occurs.

---

read

```
public int read(byte[] b)
         throws IOException
```

Reads some number of bytes from the input stream and stores them into the buffer array `b`. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.

If the length of `b` is zero, then no bytes are read and `0` is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at the end of the file, the value `-1` is returned; otherwise, at least one byte is read and stored into `b`.

The first byte read is stored into element `b[0]`, the next one into `b[1]`, and so on. The number of bytes read is, at most, equal to the length of `b`. Let *k* be the number of bytes actually read; these bytes will be stored in elements `b[0]` through `b[`*k*`-1]`, leaving elements `b[`*k*`]` through `b[b.length-1]` unaffected.

The `read(b)` method for class `InputStream` has the same effect as:

```
read(b, 0, b.length)
```

Parameters:

`b` - the buffer into which the data is read.

Returns:

the total number of bytes read into the buffer, or `-1` is there is no more data because the end of the stream has been reached.

Throws:

`IOException` - If the first byte cannot be read for any reason other than the end of the file, if the input stream has been closed, or if some other I/O error occurs.

`NullPointerException` - if `b` is `null`.

See Also:

`read(byte[], int, int)`

---

read

```
public int read(byte[] b,
                int off,
                int len)
         throws IOException
```

Reads up to `len` bytes of data from the input stream into an array of bytes. An attempt is made to read as many as `len` bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer.

This method blocks until input data is available, end of file is detected, or an exception is thrown.

If `len` is zero, then no bytes are read and `0` is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value `-1` is returned; otherwise, at least one byte is read and stored into `b`.

The first byte read is stored into element `b[off]`, the next one into `b[off+1]`, and so on. The number of bytes read is, at most, equal to `len`. Let *k* be the number of bytes actually read; these bytes will be stored in elements `b[off]` through `b[off+`*k*`-1]`, leaving elements `b[off+`*k*`]` through `b[off+len-1]` unaffected.

In every case, elements `b[0]` through `b[off]` and elements `b[off+len]` through `b[b.length-1]` are unaffected.

The `read(b, off, len)` method for class `InputStream` simply calls the method `read()` repeatedly. If the first such call results in an `IOException`, that exception is returned from the call to the `read(b, off, len)` method. If any subsequent call to `read()` results in a `IOException`, the exception is caught and treated as if it were end of file; the bytes read up to that point are stored into `b` and the number of bytes read before the exception occurred is returned. The default implementation of this method blocks until the requested amount of input data `len` has been read, end of file is detected, or an exception is thrown. Subclasses are encouraged to provide a more efficient implementation of this method.

Parameters:

`b` - the buffer into which the data is read.
`off` - the start offset in array `b` at which the data is written.
`len` - the maximum number of bytes to read.

Returns:

the total number of bytes read into the buffer, or `-1` if there is no more data because the end of the stream has been reached.

Throws:

`IOException` - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
`NullPointerException` - If `b` is `null`.
`IndexOutOfBoundsException` - If `off` is negative, `len` is negative, or `len` is greater than `b.length - off`

See Also:

`read()`

---

skip

```
public long skip(long n)
         throws IOException
```

Skips over and discards `n` bytes of data from this input stream. The `skip` method may, for a variety of reasons, end up skipping over some smaller number of bytes, possibly `0`. This may result from any of a number of conditions; reaching end of file before `n` bytes have been skipped is only one possibility. The actual number of bytes skipped is returned. If `n` is negative, no bytes are skipped.

The `skip` method of this class creates a byte array and then repeatedly reads into it until `n` bytes have been read or the end of the stream has been reached. Subclasses are encouraged to provide a more efficient implementation of this method. For instance, the implementation may depend on the ability to seek.

Parameters:

`n` - the number of bytes to be skipped.

Returns:

the actual number of bytes skipped.

Throws:

`IOException` - if the stream does not support seek, or if some other I/O error occurs.

---

available

```
public int available()
            throws IOException
```

Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream. The next invocation might be the same thread or another thread. A single read or skip of this many bytes will not block, but may read or skip fewer bytes.

Note that while some implementations of `InputStream` will return the total number of bytes in the stream, many will not. It is never correct to use the return value of this method to allocate a buffer intended to hold all data in this stream.

A subclass' implementation of this method may choose to throw an `IOException` if this input stream has been closed by invoking the `close()` method.

The `available` method for class `InputStream` always returns `0`.

This method should be overridden by subclasses.

Returns:

an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking or `0` when it reaches the end of the input stream.

Throws:

IOException - if an I/O error occurs.

---

close

```
public void close()
           throws IOException
```

Closes this input stream and releases any system resources associated with the stream.

The close method of InputStream does nothing.

Specified by:

close in interface Closeable

Throws:

IOException - if an I/O error occurs.

---

mark

```
public void mark(int readlimit)
```

Marks the current position in this input stream. A subsequent call to the reset method repositions this stream at the last marked position so that subsequent reads re-read the same bytes.

The readlimit arguments tells this input stream to allow that many bytes to be read before the mark position gets invalidated.

The general contract of mark is that, if the method markSupported returns true, the stream somehow remembers all the bytes read after the call to mark and stands ready to supply those same bytes again if and whenever the method reset is called. However, the stream is not required to remember any data at all if more than readlimit bytes are read from the stream before reset is called.

Marking a closed stream should not have any effect on the stream.

The mark method of InputStream does nothing.

Parameters:

readlimit - the maximum limit of bytes that can be read before the mark position becomes invalid.

See Also:

reset()

---

reset

```
public void reset()
```

```
throws IOException
```

Repositions this stream to the position at the time the `mark` method was last called on this input stream.

The general contract of `reset` is:

- If the method `markSupported` returns `true`, then:
  - If the method `mark` has not been called since the stream was created, or the number of bytes read from the stream since `mark` was last called is larger than the argument to `mark` at that last call, then an `IOException` might be thrown.
  - If such an `IOException` is not thrown, then the stream is reset to a state such that all the bytes read since the most recent call to `mark` (or since the start of the file, if `mark` has not been called) will be resupplied to subsequent callers of the `read` method, followed by any bytes that otherwise would have been the next input data as of the time of the call to `reset`.
- If the method `markSupported` returns `false`, then:
  - The call to `reset` may throw an `IOException`.
  - If an `IOException` is not thrown, then the stream is reset to a fixed state that depends on the particular type of the input stream and how it was created. The bytes that will be supplied to subsequent callers of the `read` method depend on the particular type of the input stream.

The method `reset` for class `InputStream` does nothing except throw an `IOException`.

Throws:

`IOException` - if this stream has not been marked or if the mark has been invalidated.

See Also:

`mark(int)`, `IOException`

---

markSupported

```
public boolean markSupported()
```

Tests if this input stream supports the `mark` and `reset` methods. Whether or not `mark` and `reset` are supported is an invariant property of a particular input stream instance. The `markSupported` method of `InputStream` returns `false`.

Returns:

`true` if this stream instance supports the mark and reset methods; `false` otherwise.

See Also:

`mark(int)`, `reset()`

Submit a bug or feature

For further API reference and developer documentation, see Java SE Developer Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.