Westerdals Oslo ACT

Skriftlig prøve
75 %

PGR101 – Objektorientert programmering 2

Tillatte hjelpemidler: ingen — Dato: 1.6.16
Vedlegg som kan være aktuelle: 13 (side 3 – 25) — Tid: 180 minutter

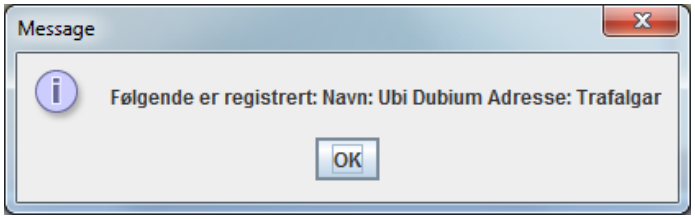
I alle oppgavene teller hvert delspørsmål likt dersom ikke annet er oppgitt.
NB! Hvis du synes noe er uklart eller at opplysninger mangler, må du gjøre egne begrunnede antagelser/forutsetninger, og løse oppgaven ut fra disse.

Oppgave 1 (30 %)

Klassen `Register` (Vedlegg 1) skal sette opp et grafisk grensesnitt for et personregister. Objekter av klassen `Person` (Vedlegg 2) brukes til å lagre opplysninger om en person.

a) (10%) Studer vedlegget, og lag en tegning som viser hvordan det grafiske grensesnittet (GUI) blir når koden blir utført.

Ved klikk på knappen `btnOk`, skal navnet og adressen som er skrevet i de to tekstfeltene `txtName` og `txtAdress` (hvis det står noe der) brukes til å opprette et objekt av klassen `Person`. Dette objektet skal legges til i listen `persons`. Tekstfeltene skal så tømmes, og det skal vises en melding om at en registrering er gjort:



b) (10%) Skriv koden som skal legges inn i metoden `actionPerformed` slik at effekten av klikk på `btnOk` blir som beskrevet over.

Ved klikk på knappen `btnShow`, skal innholdet i listen `persons` vises i tekstområdet `txaDisplay`, og ved klikk på knappen `btnExit`, skal applikasjonen avsluttes.

c) (10%) Skriv koden som må legges til i metoden `actionPerformed` slik at effekten av klikk på `btnShow` og `btnExit` blir som beskrevet over.

Oppgave 2 (20 %)

Anta at du skriver en klasse `Student`, og at du ikke utstyrer klassen med en `toString`-metode. Klassen har ingen deklarert superklasse. Betrakt følgende kodelinjer:

```
Student stud = new Student();
String s = stud.toString();
System.out.println(s);
```

a) (10%) Vil disse linjene kompilere uten feilmelding? Beskriv hva utskriften vil vise hvis kompileringen går bra og kodelinjene blir utført. Forklar hvorfor resultatet blir som det blir.

Studer klassene gitt i Vedlegg 3.

b) (10%) Hva blir skrevet ut når `method` i klassen `Client2` blir kalt?

Oppgave 3 (30 %)

Tekstfilen `results.txt` inneholder resultatene fra en prøve i fagene Prog og Tek. Det som er oppgitt for hver kandidat er fag, studentnummer og karakter – på hver sin linje. Et eksempel på utskrift av en slik fil (et utdrag) er vist under til venstre.
En metode `mainMethod` (se figuren under til høyre) kaller metodene `readResults` og `saveResults` med bl.a. `results.txt` som argument.

```
Prog
710002
C
Tek
710006
B
Prog
710005
B
Prog
710010
A
Tek
710002
D
```

```
public void mainMethod() {
    readResults("results.txt");
    saveResults("results.txt", "prog.txt", "tek.txt");
}
```

a) (15%) Skriv metoden `readResults`. Metoden har et filnavn som parameter, og den leser filen angitt av parameteren. Metoden skal lage en utskrift som den som er vist ovenfor til venstre.

b) (15%) Skriv metoden `saveResults`. Metoden har tre filnavn som parametere, og den skal lese filen angitt ved parameter 1 (se figur over til høyre), og så plukke ut og lagre Prog-resultatene til filen angitt av parameter 2, og Tek-resultatene skal lagres til filen angitt av parameter 3.
Metoden skal i tillegg telle opp og skrive ut i terminalvinduet hvor mange kandidater som har avlagt eksamen i hvert av de to fagene.

Oppgave 4 (20%)

a) (10%) Gi et eksempel på en `checked` og en `unchecked exception`. Hva er forskjellen på disse?

b) (10%) På hvilke måter kan en metode `readData`, som leser data fra en fil, håndtere en `checked exception`?

--- Slutt på oppgavesettet ---

Vedlegg

Vedlegg 1 Klassen `Register`

```
public class Register extends JFrame implements ActionListener {
    private JButton btnOk;
    private JButton btnShow;
    private JButton btnCancel;
    private JButton btnExit;
    private JTextField txtName;
    private JTextField txtAdress;
    private JTextArea txaDisplay;
    private ArrayList<Person> persons;

    public Register() {
        setTitle("REGISTRERING");
        persons = new ArrayList<Person>();

        btnOk = new JButton("OK");
        btnShow = new JButton("Vis innhold");
        btnCancel = new JButton("Avbryt");
        btnExit = new JButton("Avslutt");
        btnOk.addActionListener(this);
        btnShow.addActionListener(this);
        btnCancel.addActionListener(this);
        btnExit.addActionListener(this);

        txaDisplay = new JTextArea(10, 10);
        txtName = new JTextField(15);
        txtAdress = new JTextField(15);

        JPanel pnlNorth = new JPanel();
        pnlNorth.setLayout(new GridLayout(2, 2));
        pnlNorth.add(new JLabel("Navn"));
        pnlNorth.add(txtName);
        pnlNorth.add(new JLabel("Adresse"));
        pnlNorth.add(txtAdress);

        JPanel pnlSouth = new JPanel();
        pnlSouth.setLayout(new GridLayout(1, 4));
        pnlSouth.add(btnOk);
        pnlSouth.add(btnShow);
        pnlSouth.add(btnCancel);
        pnlSouth.add(btnExit);

        add(pnlNorth, BorderLayout.NORTH);
        add(txaDisplay,BorderLayout.CENTER);
        add(pnlSouth, BorderLayout.SOUTH);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {

    }

}
```

Vedlegg 2 Klassen `Person`

```
public class Person {
   private String name;
   private String adress;

   public Person() {
      this("", "");
   }

   public Person(String name, String adress) {
      setName(name);
      setAdress(adress);
   }

   public String getName() {
      return name;
   }

   public String getAdress() {
      return adress;
   }

   public void setName(String name) {
      this.name = name;
   }

   public void setAdress(String adress) {
      this.adress = adress;
   }

   public boolean equals(Object obj) {
      if (!(obj instanceof Person)) return false;
      if (obj == this) return true;
      Person p = (Person) obj;
      return getName().equals(p.getName()) && getAdress().equals(p.getAdress());
   }

   public String toString() {
      return getName() + " " + getAdress();
   }
}
```

Vedlegg 3

```
public class SuperClass {
    public SuperClass() {

    }

    public void method1() {
        System.out.println("Metode 1 i SuperClass");
    }

    public void method2() {
        System.out.println("Metode 2 i SuperClass");
    }
}
```

```
public class SubClass1 extends SuperClass {
    public SubClass1() {

    }

    public void method1() {
        System.out.println("Metode 1 i SubClass1");
    }
}
```

```
public class SubClass2 extends SubClass1 {
    public SubClass2() {

    }

    public void method1() {
        System.out.println("Metode 1 i SubClass2");
    }

    public void method2() {
        super.method1();
    }
}
```

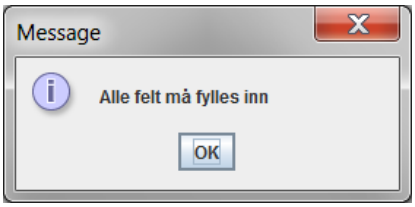
```
public class Client2 {
    public void method() {
        SuperClass sup = new SuperClass();
        SuperClass sub1 = new SubClass1();
        SuperClass sub2 = new SubClass2();

        sup.method1();
        sup.method2();
        sub1.method1();
        sub1.method2();
        sub2.method1();
        sub2.method2();
    }
}
```

Vedlegg 4 `JOptionPane` sine static metoder for å lage standard dialoger

---

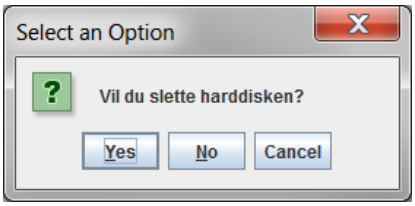
`showMessageDialog(`***<parent>***`,` ***<message>***`)`
Viser en melding med en OK-knapp.



`showConfirmDialog(`***<parent>***`,` ***<message>***`)`
Viser en melding og knapper for tre valg:
Yes, No, Cancel

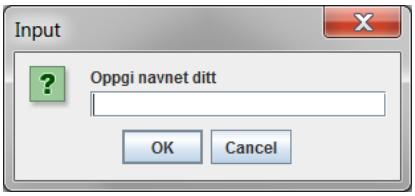
Returnerer brukerens valg som en `int` med en av følgende verdier:

`JOptionPane.YES_OPTION`
`JOptionPane.NO_OPTION`
`JOptionPane.CANCEL_OPTION`



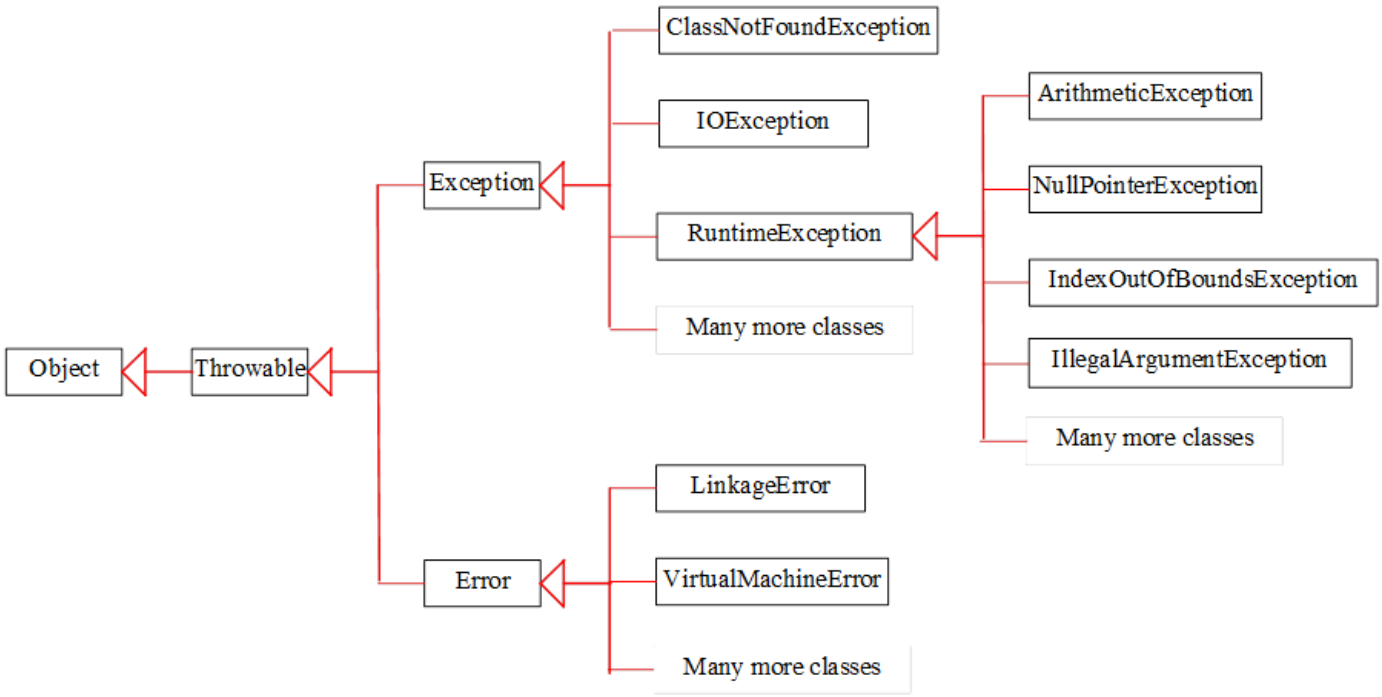
`showInputDialog(`***<parent>***`,` ***<message>***`)`
Viser en melding og et tekstfelt for input.

Returnerer brukers verdi som en `String`.



"parent" angir, for hver dialog, hvilket vindu dialogen skal ligge midt oppå.

Vedlegg 5 Java Exception Hierarki (utdrag.)



Vedlegg 6 Klassen `JTextArea`

javax.swing

Class JTextArea

java.lang.Object
  java.awt.Component
    java.awt.Container
      javax.swing.JComponent
        javax.swing.text.JTextComponent
          javax.swing.JTextArea

All Implemented Interfaces:

ImageObserver, MenuContainer, Serializable, Accessible, Scrollable

---

```
public class JTextArea
extends JTextComponent
```

A `JTextArea` is a multi-line area that displays plain text. It is intended to be a lightweight component that provides source compatibility with the`java.awt.TextArea` class where it can reasonably do so. You can find information and examples of using all the text components in Using Text Components, a section in *The Java Tutorial*.

This component has capabilities not found in the `java.awt.TextArea` class. The superclass should be consulted for additional capabilities. Alternative multi-line text classes with more capabilities are `JTextPane` and `JEditorPane`.

The `java.awt.TextArea` internally handles scrolling. `JTextArea` is different in that it doesn't manage scrolling, but implements the swing `Scrollable` interface. This allows it to be placed inside a `JScrollPane` if scrolling behavior is desired, and used directly if scrolling is not desired.

The `java.awt.TextArea` has the ability to do line wrapping. This was controlled by the horizontal scrolling policy. Since scrolling is not done by `JTextArea`directly, backward compatibility must be provided another way. `JTextArea` has a bound property for line wrapping that controls whether or not it will wrap lines. By default, the line wrapping property is set to false (not wrapped).

`java.awt.TextArea` has two properties `rows` and `columns` that are used to determine the preferred size. `JTextArea` uses these properties to indicate the preferred size of the viewport when placed inside a `JScrollPane` to match the functionality provided by `java.awt.TextArea`. `JTextArea` has a preferred size of what is needed to display all of the text, so that it functions properly inside of a `JScrollPane`. If the value for `rows` or `columns` is equal to zero, the preferred size along that axis is used for the viewport preferred size along the same axis.

The `java.awt.TextArea` could be monitored for changes by adding a `TextListener` for `TextEvent`s. In the `JTextComponent` based components, changes are broadcasted from the model via a `DocumentEvent` to `DocumentListeners`. The `DocumentEvent` gives the location of the change and the kind of change if desired. The code fragment might look something like:

```
DocumentListener myListener = ??;
JTextArea myArea = ??;
myArea.getDocument().addDocumentListener(myListener);
```

Newlines

For a discussion on how newlines are handled, see `DefaultEditorKit`.

Warning: Swing is not thread safe. For more information see Swing's Threading Policy.

Warning: Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications running the same version of Swing. As of 1.4, support for long term storage of all JavaBeans™ has been added to the `java.beans`package. Please see `XMLEncoder`.

See Also:

`JTextPane`, `JEditorPane`

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
protected class	**`JTextArea.AccessibleJTextArea`** This class implements accessibility support for the `JTextArea` class.

Nested classes/interfaces inherited from class javax.swing.text.JTextComponent

`JTextComponent.AccessibleJTextComponent`, `JTextComponent.DropLocation`, `JTextComponent.KeyBinding`

Nested classes/interfaces inherited from class javax.swing.JComponent

`JComponent.AccessibleJComponent`

Nested classes/interfaces inherited from class java.awt.Container

`Container.AccessibleAWTContainer`

Nested classes/interfaces inherited from class java.awt.Component

`Component.AccessibleAWTComponent`, `Component.BaselineResizeBehavior`, `Component.BltBufferStrategy`, `Component.FlipBufferStrategy`

Field Summary

Fields inherited from class javax.swing.text.JTextComponent

`DEFAULT_KEYMAP`, `FOCUS_ACCELERATOR_KEY`

Fields inherited from class javax.swing.JComponent

`listenerList`, `TOOL_TIP_TEXT_KEY`, `ui`, `UNDEFINED_CONDITION`, `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT`, `WHEN_FOCUSED`, `WHEN_IN_FOCUSED_WINDOW`

Fields inherited from class java.awt.Component

accessibleContext, BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface java.awt.image.ImageObserver

ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

Constructors

Constructor and Description
JTextArea() Constructs a new TextArea.
JTextArea(**Document** doc) Constructs a new JTextArea with the given document model, and defaults for all of the other arguments (null, 0, 0).
JTextArea(**Document** doc, **String** text, int rows, int columns) Constructs a new JTextArea with the specified number of rows and columns, and the given model.
JTextArea(int rows, int columns) Constructs a new empty TextArea with the specified number of rows and columns.
JTextArea(**String** text) Constructs a new TextArea with the specified text displayed.
JTextArea(**String** text, int rows, int columns) Constructs a new TextArea with the specified text and number of rows and columns.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
void	**append**(**String** str) Appends the given text to the end of the document.
protected **Document**	**createDefaultModel**() Creates the default implementation of the model to be used at construction if one isn't explicitly given.
AccessibleContext	**getAccessibleContext**() Gets the AccessibleContext associated with this JTextArea.
int	**getColumns**() Returns the number of columns in the TextArea.
protected int	**getColumnWidth**() Gets column width.
int	**getLineCount**() Determines the number of lines contained in the area.
int	**getLineEndOffset**(int line) Determines the offset of the end of the given line.
int	**getLineOfOffset**(int offset) Translates an offset into the components text to a line number.
int	**getLineStartOffset**(int line) Determines the offset of the start of the given line.
boolean	**getLineWrap**() Gets the line-wrapping policy of the text area.
Dimension	**getPreferredScrollableViewportSize**() Returns the preferred size of the viewport if this component is embedded in a JScrollPane.
Dimension	**getPreferredSize**() Returns the preferred size of the TextArea.
protected int	**getRowHeight**() Defines the meaning of the height of a row.
int	**getRows**() Returns the number of rows in the TextArea.
boolean	**getScrollableTracksViewportWidth**() Returns true if a viewport should always force the width of this Scrollable to match the width of the viewport.
int	**getScrollableUnitIncrement**(**Rectangle** visibleRect, int orientation, int direction)

	Components that display logical rows or columns should compute the scroll increment that will completely expose one new row or column, depending on the value of orientation.
int	**getTabSize**() Gets the number of characters used to expand tabs.
String	**getUIClassID**() Returns the class ID for the UI.
boolean	**getWrapStyleWord**() Gets the style of wrapping used if the text area is wrapping lines.
void	**insert**(**String** str, int pos) Inserts the specified text at the specified position.
protected **String**	**paramString**() Returns a string representation of this JTextArea.
void	**replaceRange**(**String** str, int start, int end) Replaces text from the indicated start to end position with the new text specified.
void	**setColumns**(int columns) Sets the number of columns for this TextArea.
void	**setFont**(**Font** f) Sets the current font.
void	**setLineWrap**(boolean wrap) Sets the line-wrapping policy of the text area.
void	**setRows**(int rows) Sets the number of rows for this TextArea.
void	**setTabSize**(int size) Sets the number of characters to expand tabs to.
void	**setWrapStyleWord**(boolean word) Sets the style of wrapping used if the text area is wrapping lines.

Vedlegg 7 Klassen `ActionEvent`

java.awt.event

Class ActionEvent

java.lang.Object
  java.util.EventObject
    java.awt.AWTEvent
      java.awt.event.ActionEvent

All Implemented Interfaces:

Serializable

`public class` **`ActionEvent`**
`extends` AWTEvent

A semantic event which indicates that a component-defined action occurred. This high-level event is generated by a component (such as a `Button`) when the component-specific action occurs (such as being pressed). The event is passed to every `ActionListener` object that registered to receive such events using the component's `addActionListener` method.

Note: To invoke an `ActionEvent` on a `Button` using the keyboard, use the Space bar.

The object that implements the `ActionListener` interface gets this `ActionEvent` when the event occurs. The listener is therefore spared the details of processing individual mouse movements and mouse clicks, and can instead process a "meaningful" (semantic) event like "button pressed".

An unspecified behavior will be caused if the `id` parameter of any particular `ActionEvent` instance is not in the range from `ACTION_FIRST` to `ACTION_LAST`.

Since:

1.1

See Also:

`ActionListener`, Tutorial: How to Write an Action Listener, Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
static int	**ACTION_FIRST** The first number in the range of ids used for action events.
static int	**ACTION_LAST** The last number in the range of ids used for action events.

static int	**ACTION_PERFORMED** This event id indicates that a meaningful action occurred.
static int	**ALT_MASK** The alt modifier.
static int	**CTRL_MASK** The control modifier.
static int	**META_MASK** The meta modifier.
static int	**SHIFT_MASK** The shift modifier.

Fields inherited from class java.awt.AWTEvent

ACTION_EVENT_MASK, ADJUSTMENT_EVENT_MASK, COMPONENT_EVENT_MASK, consumed, CONTAINER_EVENT_MASK, FOCUS_EVENT_MASK, HIERARCHY_BOUNDS_EVENT_MASK, HIERARCHY_EVENT_MASK, id, INPUT_METHOD_EVENT_MASK, INVOCATION_EVENT_MASK, ITEM_EVENT_MASK, KEY_EVENT_MASK, MOUSE_EVENT_MASK, MOUSE_MOTION_EVENT_MASK, MOUSE_WHEEL_EVENT_MASK, PAINT_EVENT_MASK, RESERVED_ID_MAX, TEXT_EVENT_MASK, WINDOW_EVENT_MASK, WINDOW_FOCUS_EVENT_MASK, WINDOW_STATE_EVENT_MASK

Fields inherited from class java.util.EventObject

source

Constructor Summary

Constructors

Constructor and Description
ActionEvent(Object source, int id, String command) Constructs an ActionEvent object.
ActionEvent(Object source, int id, String command, int modifiers) Constructs an ActionEvent object with modifier keys.
ActionEvent(Object source, int id, String command, long when, int modifiers) Constructs an ActionEvent object with the specified modifier keys and timestamp.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
String	**getActionCommand**() Returns the command string associated with this action.
int	**getModifiers**() Returns the modifier keys held down during this action event.
long	**getWhen**() Returns the timestamp of when this event occurred.
String	**paramString**() Returns a parameter string identifying this action event.

Vedlegg 8 Klassen Scanner

java.util
Class Scanner
java.lang.Object

java.util.Scanner
All Implemented Interfaces:
Closeable, AutoCloseable, Iterator<String>

```
public final class Scanner
extends Object
implements Iterator<String>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressions.
A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.
For example, this code allows a user to read a number from System.in:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

As another example, this code allows long types to be assigned from entries in a file myNumbers:

```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
    long aLong = sc.nextLong();
}
```

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

prints the following output:

```
1
2
red
blue
```

The same output can be generated with this code, which uses a regular expression to parse all four tokens at once:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");
MatchResult result = s.match();
for (int i=1; i<=result.groupCount(); i++)
    System.out.println(result.group(i));
s.close();
```

The default whitespace delimiter used by a scanner is as recognized by `Character.isWhitespace`. The `reset()` method will reset the value of the scanner's delimiter to the default whitespace delimiter regardless of whether it was previously changed.
A scanning operation may block waiting for input.
The `next()` and `hasNext()` methods and their primitive-type companion methods (such as `nextInt()` and `hasNextInt()`) first skip any input that matches the delimiter pattern, and then attempt to return the next token. Both `hasNext` and `next` methods may block waiting for further input. Whether a `hasNext` method blocks has no connection to whether or not its associated `next` method will block.
The `findInLine(java.lang.String)`, `findWithinHorizon(java.lang.String, int)`, and `skip(java.util.regex.Pattern)` methods operate independently of the delimiter pattern. These methods will attempt to match the specified pattern with no regard to delimiters in the input and thus can be used in special circumstances where delimiters are not relevant. These methods may block waiting for more input.
When a scanner throws an `InputMismatchException`, the scanner will not pass the token that caused the exception, so that it may be retrieved or skipped via some other method.
Depending upon the type of delimiting pattern, empty tokens may be returned. For example, the pattern `"\\s+"` will return no empty tokens since it matches multiple instances of the delimiter. The delimiting pattern `"\\s"` could return empty tokens since it only passes one space at a time.
A scanner can read text from any object which implements the `Readable` interface. If an invocation of the underlying readable's`Readable.read(java.nio.CharBuffer)` method throws an `IOException` then the scanner assumes that the end of the input has been reached. The most recent`IOException` thrown by the underlying readable can be retrieved via the `ioException()` method.
When a `Scanner` is closed, it will close its input source if the source implements the `Closeable` interface.
A `Scanner` is not safe for multithreaded use without external synchronization.
Unless otherwise mentioned, passing a `null` parameter into any method of a `Scanner` will cause a `NullPointerException` to be thrown.
A scanner will default to interpreting numbers as decimal unless a different radix has been set by using the `useRadix(int)` method.
The `reset()` method will reset the value of the scanner's radix to `10` regardless of whether it was previously changed.
Localized numbers
An instance of this class is capable of scanning numbers in the standard formats as well as in the formats of the scanner's locale. A scanner's initial locale is the value returned by the `Locale.getDefault(Locale.Category.FORMAT)` method; it may be changed via the `useLocale(java.util.Locale)` method. The `reset()` method will reset the value of the scanner's locale to the initial locale regardless of whether it was previously changed.
The localized formats are defined in terms of the following parameters, which for a particular locale are taken from that locale's `DecimalFormat` object, `df`, and its and `DecimalFormatSymbols` object, `dfs`.

LocalGroupSeparator
: The character used to separate thousands groups, *i.e.*, dfs.getGroupingSeparator()

LocalDecimalSeparator
: The character used for the decimal point, *i.e.*, dfs.getDecimalSeparator()

LocalPositivePrefix
: The string that appears before a positive number (may be empty), *i.e.*, df.getPositivePrefix()

LocalPositiveSuffix
: The string that appears after a positive number (may be empty), *i.e.*, df.getPositiveSuffix()

LocalNegativePrefix
: The string that appears before a negative number (may be empty), *i.e.*, df.getNegativePrefix()

LocalNegativeSuffix
: The string that appears after a negative number (may be empty), *i.e.*, df.getNegativeSuffix()

LocalNaN
: The string that represents not-a-number for floating-point values, *i.e.*, dfs.getNaN()

LocalInfinity
: The string that represents infinity for floating-point values, *i.e.*, dfs.getInfinity()

Number syntax
The strings that can be parsed as numbers by an instance of this class are specified in terms of the following regular-expression grammar, where Rmax is the highest digit in the radix being used (for example, Rmax is 9 in base 10).

NonAsciiDigit:
: A non-ASCII character c for which `Character.isDigit(c)` returns true

NonoDigit:
```
[1-Rmax] | NonASCIIDigit
```
Digit:
```
[0-Rmax] | NonASCIIDigit
```
GroupedNumeral:
```
( Non0Digit Digit? Digit?
    ( LocalGroupSeparator Digit Digit Digit )+ )
```
Numeral:
```
( ( Digit+ ) | GroupedNumeral )
```
Integer:
```
( [-+]? ( Numeral ) )
| LocalPositivePrefix Numeral LocalPositiveSuffix
| LocalNegativePrefix Numeral LocalNegativeSuffix
```
DecimalNumeral:
```
Numeral
| Numeral LocalDecimalSeparator Digit*
| LocalDecimalSeparator Digit+
```
Exponent:
```
( [eE] [+-]? Digit+ )
```
Decimal:
```
( [-+]? DecimalNumeral Exponent? )
| LocalPositivePrefix DecimalNumeral LocalPositiveSuffix Exponent?
| LocalNegativePrefix DecimalNumeral LocalNegativeSuffix Exponent?
```
HexFloat:
```
[-+]? 0[xX][0-9a-fA-F]*\.[0-9a-fA-F]+ ([pP][-+]?[0-9]+)?
```
NonNumber:
```
NaN | LocalNan | Infinity | LocalInfinity
```
SignedNonNumber:
```
( [-+]? NonNumber )
| LocalPositivePrefix NonNumber LocalPositiveSuffix
| LocalNegativePrefix NonNumber LocalNegativeSuffix
```
Float:
```
Decimal | HexFloat | SignedNonNumber
```
Whitespace is not significant in the above regular expressions.

Since:

1.5

Constructor Summary

Constructors

Constructor and Description
`Scanner(File source)` Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(File source, String charsetName)` Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(InputStream source)` Constructs a new `Scanner` that produces values scanned from the specified input stream.
`Scanner(InputStream source, String charsetName)` Constructs a new `Scanner` that produces values scanned from the specified input stream.
`Scanner(Path source)` Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(Path source, String charsetName)` Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(Readable source)` Constructs a new `Scanner` that produces values scanned from the specified source.
`Scanner(ReadableByteChannel source)` Constructs a new `Scanner` that produces values scanned from the specified channel.
`Scanner(ReadableByteChannel source, String charsetName)` Constructs a new `Scanner` that produces values scanned from the specified channel.
`Scanner(String source)` Constructs a new `Scanner` that produces values scanned from the specified string.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
`void`	`close()` Closes this scanner.
`Pattern`	`delimiter()`

	Returns the `Pattern` this `Scanner` is currently using to match delimiters.
String	**findInLine**(**Pattern** pattern) Attempts to find the next occurrence of the specified pattern ignoring delimiters.
String	**findInLine**(**String** pattern) Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
String	**findWithinHorizon**(**Pattern** pattern, int horizon) Attempts to find the next occurrence of the specified pattern.
String	**findWithinHorizon**(**String** pattern, int horizon) Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
boolean	**hasNext**() Returns true if this scanner has another token in its input.
boolean	**hasNext**(**Pattern** pattern) Returns true if the next complete token matches the specified pattern.
boolean	**hasNext**(**String** pattern) Returns true if the next token matches the pattern constructed from the specified string.
boolean	**hasNextBigDecimal**() Returns true if the next token in this scanner's input can be interpreted as a `BigDecimal` using the**nextBigDecimal()** method.
boolean	**hasNextBigInteger**() Returns true if the next token in this scanner's input can be interpreted as a `BigInteger` in the default radix using the **nextBigInteger()** method.
boolean	**hasNextBigInteger**(int radix) Returns true if the next token in this scanner's input can be interpreted as a `BigInteger` in the specified radix using the **nextBigInteger()** method.
boolean	**hasNextBoolean**() Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false".
boolean	**hasNextByte**() Returns true if the next token in this scanner's input can be interpreted as a byte value in the default radix using the**nextByte()** method.
boolean	**hasNextByte**(int radix) Returns true if the next token in this scanner's input can be interpreted as a byte value in the specified radix using the **nextByte()** method.
boolean	**hasNextDouble**() Returns true if the next token in this scanner's input can be interpreted as a double value using the **nextDouble()** method.
boolean	**hasNextFloat**() Returns true if the next token in this scanner's input can be interpreted as a float value using the **nextFloat()** method.
boolean	**hasNextInt**() Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the**nextInt()** method.
boolean	**hasNextInt**(int radix) Returns true if the next token in this scanner's input can be interpreted as an int value in the specified radix using the **nextInt()** method.
boolean	**hasNextLine**() Returns true if there is another line in the input of this scanner.
boolean	**hasNextLong**() Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the**nextLong()** method.
boolean	**hasNextLong**(int radix) Returns true if the next token in this scanner's input can be interpreted as a long value in the specified radix using the **nextLong()** method.
boolean	**hasNextShort**()

	Returns true if the next token in this scanner's input can be interpreted as a short value in the default radix using the **nextShort()** method.
boolean	**hasNextShort**(int radix) Returns true if the next token in this scanner's input can be interpreted as a short value in the specified radix using the **nextShort()** method.
IOException	**ioException**() Returns the IOException last thrown by this Scanner's underlying Readable.
Locale	**locale**() Returns this scanner's locale.
MatchResult	**match**() Returns the match result of the last scanning operation performed by this scanner.
String	**next**() Finds and returns the next complete token from this scanner.
String	**next**(**Pattern** pattern) Returns the next token if it matches the specified pattern.
String	**next**(**String** pattern) Returns the next token if it matches the pattern constructed from the specified string.
BigDecimal	**nextBigDecimal**() Scans the next token of the input as a **BigDecimal**.
BigInteger	**nextBigInteger**() Scans the next token of the input as a **BigInteger**.
BigInteger	**nextBigInteger**(int radix) Scans the next token of the input as a **BigInteger**.
boolean	**nextBoolean**() Scans the next token of the input into a boolean value and returns that value.
byte	**nextByte**() Scans the next token of the input as a byte.
byte	**nextByte**(int radix) Scans the next token of the input as a byte.
double	**nextDouble**() Scans the next token of the input as a double.
float	**nextFloat**() Scans the next token of the input as a float.
int	**nextInt**() Scans the next token of the input as an int.
int	**nextInt**(int radix) Scans the next token of the input as an int.
String	**nextLine**() Advances this scanner past the current line and returns the input that was skipped.
long	**nextLong**() Scans the next token of the input as a long.
long	**nextLong**(int radix) Scans the next token of the input as a long.
short	**nextShort**() Scans the next token of the input as a short.
short	**nextShort**(int radix) Scans the next token of the input as a short.
int	**radix**() Returns this scanner's default radix.
void	**remove**() The remove operation is not supported by this implementation of Iterator.

Scanner	reset() Resets this scanner.
Scanner	skip(Pattern pattern) Skips input that matches the specified pattern, ignoring delimiters.
Scanner	skip(String pattern) Skips input that matches a pattern constructed from the specified string.
String	toString() Returns the string representation of this Scanner.
Scanner	useDelimiter(Pattern pattern) Sets this scanner's delimiting pattern to the specified pattern.
Scanner	useDelimiter(String pattern) Sets this scanner's delimiting pattern to a pattern constructed from the specified String.
Scanner	useLocale(Locale locale) Sets this scanner's locale to the specified locale.
Scanner	useRadix(int radix) Sets this scanner's default radix to the specified radix.

Vedlegg 9 Klassen `File`

java.io
Class File
java.lang.Object
java.io.File
All Implemented Interfaces:
`Serializable`, `Comparable<File>`

```
public class File
extends Object
implements Serializable, Comparable<File>
```

An abstract representation of file and directory pathnames.
User interfaces and operating systems use system-dependent *pathname strings* to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames. An *abstract pathname* has two components:

1. An optional system-dependent *prefix* string, such as a disk-drive specifier, `"/"` for the UNIX root directory, or `"\\\\"` for a Microsoft Windows UNC pathname, and
2. A sequence of zero or more string *names*.

The first name in an abstract pathname may be a directory name or, in the case of Microsoft Windows UNC pathnames, a hostname. Each subsequent name in an abstract pathname denotes a directory; the last name may denote either a directory or a file. The *empty* abstract pathname has no prefix and an empty name sequence.
The conversion of a pathname string to or from an abstract pathname is inherently system-dependent. When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default *separator character*. The default name-separator character is defined by the system property `file.separator`, and is made available in the public static fields `separator` and `separatorChar` of this class. When a pathname string is converted into an abstract pathname, the names within it may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying system.
A pathname, whether abstract or in string form, may be either *absolute* or *relative*. An absolute pathname is complete in that no other information is required in order to locate the file that it denotes. A relative pathname, in contrast, must be interpreted in terms of information taken from some other pathname. By default the classes in the `java.io` package always resolve relative pathnames against the current user directory. This directory is named by the system property`user.dir`, and is typically the directory in which the Java virtual machine was invoked.
The *parent* of an abstract pathname may be obtained by invoking the `getParent()` method of this class and consists of the pathname's prefix and each name in the pathname's name sequence except for the last. Each directory's absolute pathname is an ancestor of any `File` object with an absolute abstract pathname which begins with the directory's absolute pathname. For example, the directory denoted by the abstract pathname `"/usr"` is an ancestor of the directory denoted by the pathname `"/usr/local/bin"`.
The prefix concept is used to handle root directories on UNIX platforms, and drive specifiers, root directories and UNC pathnames on Microsoft Windows platforms, as follows:

- For UNIX platforms, the prefix of an absolute pathname is always `"/"`. Relative pathnames have no prefix. The abstract pathname denoting the root directory has the prefix `"/"` and an empty name sequence.
- For Microsoft Windows platforms, the prefix of a pathname that contains a drive specifier consists of the drive letter followed by `":"` and possibly followed by `"\\"` if the pathname is absolute. The prefix of a UNC pathname is `"\\\\"`; the hostname and the share name are the first two names in the name sequence. A relative pathname that does not specify a drive has no prefix.

Instances of this class may or may not denote an actual file-system object such as a file or a directory. If it does denote such an object then that object resides in a*partition*. A partition is an operating system-specific portion of storage for a file system. A single storage device (e.g. a physical disk-drive, flash memory, CD-ROM) may contain multiple partitions. The object, if any, will reside on the partition named by some ancestor of the absolute form of this pathname.
A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as *access permissions*. The file system may have multiple sets of access permissions on a single object. For example, one set may apply to the object's *owner*, and another may apply to all other users. The access permissions on an object may cause some methods in this class to fail.
Instances of the `File` class are immutable; that is, once created, the abstract pathname represented by a `File` object will never change.
Interoperability with `java.nio.file` package
The `java.nio.file` package defines interfaces and classes for the Java virtual machine to access files, file attributes, and file systems. This API may be used to overcome many of the limitations of the `java.io.File` class. The `toPath` method may be used to obtain a `Path` that uses the abstract path represented by a `File`object to locate a file. The resulting `Path` may be used with the `Files` class to provide more efficient and extensive access to additional file operations, file attributes, and I/O exceptions to help diagnose errors when an operation on a file fails.
Since:
`JDK1.0`
See Also:
`Serialized Form`

Field Summary

Fields

Modifier and Type	Field and Description
`static String`	**`pathSeparator`** The system-dependent path-separator character, represented as a string for convenience.
`static char`	**`pathSeparatorChar`** The system-dependent path-separator character.
`static String`	**`separator`** The system-dependent default name-separator character, represented as a string for convenience.
`static char`	**`separatorChar`** The system-dependent default name-separator character.

Constructor Summary

Constructors

Constructor and Description
File(**File** parent, **String** child) Creates a new File instance from a parent abstract pathname and a child pathname string.
File(**String** pathname) Creates a new File instance by converting the given pathname string into an abstract pathname.
File(**String** parent, **String** child) Creates a new File instance from a parent pathname string and a child pathname string.
File(**URI** uri) Creates a new File instance by converting the given file: URI into an abstract pathname.

Method Summary

odsInstance MethodsConcrete MethodsDeprecated Methods

Modifier and Type	Method and Description
boolean	**canExecute**() Tests whether the application can execute the file denoted by this abstract pathname.
boolean	**canRead**() Tests whether the application can read the file denoted by this abstract pathname.
boolean	**canWrite**() Tests whether the application can modify the file denoted by this abstract pathname.
int	**compareTo**(**File** pathname) Compares two abstract pathnames lexicographically.
boolean	**createNewFile**() Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
static **File**	**createTempFile**(**String** prefix, **String** suffix) Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
static **File**	**createTempFile**(**String** prefix, **String** suffix, **File** directory) Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
boolean	**delete**() Deletes the file or directory denoted by this abstract pathname.
void	**deleteOnExit**() Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
boolean	**equals**(**Object** obj) Tests this abstract pathname for equality with the given object.
boolean	**exists**() Tests whether the file or directory denoted by this abstract pathname exists.
File	**getAbsoluteFile**() Returns the absolute form of this abstract pathname.
String	**getAbsolutePath**() Returns the absolute pathname string of this abstract pathname.
File	**getCanonicalFile**() Returns the canonical form of this abstract pathname.
String	**getCanonicalPath**() Returns the canonical pathname string of this abstract pathname.
long	**getFreeSpace**() Returns the number of unallocated bytes in the partition **named** by this abstract path name.
String	**getName**() Returns the name of the file or directory denoted by this abstract pathname.
String	**getParent**() Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
File	**getParentFile**()

	Returns the abstract pathname of this abstract pathname's parent, or `null` if this pathname does not name a parent directory.
String	**getPath**() Converts this abstract pathname into a pathname string.
long	**getTotalSpace**() Returns the size of the partition **named** by this abstract pathname.
long	**getUsableSpace**() Returns the number of bytes available to this virtual machine on the partition **named** by this abstract pathname.
int	**hashCode**() Computes a hash code for this abstract pathname.
boolean	**isAbsolute**() Tests whether this abstract pathname is absolute.
boolean	**isDirectory**() Tests whether the file denoted by this abstract pathname is a directory.
boolean	**isFile**() Tests whether the file denoted by this abstract pathname is a normal file.
boolean	**isHidden**() Tests whether the file named by this abstract pathname is a hidden file.
long	**lastModified**() Returns the time that the file denoted by this abstract pathname was last modified.
long	**length**() Returns the length of the file denoted by this abstract pathname.
String[]	**list**() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
String[]	**list**(**FilenameFilter** filter) Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	**listFiles**() Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
File[]	**listFiles**(**FileFilter** filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	**listFiles**(**FilenameFilter** filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
static **File**[]	**listRoots**() List the available filesystem roots.
boolean	**mkdir**() Creates the directory named by this abstract pathname.
boolean	**mkdirs**() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.
boolean	**renameTo**(**File** dest) Renames the file denoted by this abstract pathname.
boolean	**setExecutable**(boolean executable) A convenience method to set the owner's execute permission for this abstract pathname.
boolean	**setExecutable**(boolean executable, boolean ownerOnly) Sets the owner's or everybody's execute permission for this abstract pathname.
boolean	**setLastModified**(long time) Sets the last-modified time of the file or directory named by this abstract pathname.
boolean	**setReadable**(boolean readable) A convenience method to set the owner's read permission for this abstract pathname.
boolean	**setReadable**(boolean readable, boolean ownerOnly) Sets the owner's or everybody's read permission for this abstract pathname.
boolean	**setReadOnly**()

	Marks the file or directory named by this abstract pathname so that only read operations are allowed.
boolean	**setWritable**(boolean writable) A convenience method to set the owner's write permission for this abstract pathname.
boolean	**setWritable**(boolean writable, boolean ownerOnly) Sets the owner's or everybody's write permission for this abstract pathname.
Path	**toPath**() Returns a **java.nio.file.Path** object constructed from the this abstract path.
String	**toString**() Returns the pathname string of this abstract pathname.
URI	**toURI**() Constructs a `file:` URI that represents this abstract pathname.
URL	**toURL**() **Deprecated.** This method does not automatically escape characters that are illegal in URLs. It is recommended that new code convert an abstract pathname into a URL by first converting it into a URI, via the **toURI** method, and then converting the URI into a URL via the **URI.toURL** method.

Vedlegg 10 Klassen `PrintStream`

java.io

Class PrintStream

java.lang.Object
  java.io.OutputStream
    java.io.FilterOutputStream
      java.io.PrintStream

All Implemented Interfaces:

Closeable, Flushable, Appendable, AutoCloseable

Direct Known Subclasses:

LogStream

```
public class PrintStream
extends FilterOutputStream
implements Appendable, Closeable
```

A `PrintStream` adds functionality to another output stream, namely the ability to print representations of various data values conveniently. Two other features are provided as well. Unlike other output streams, a `PrintStream` never throws an `IOException`; instead, exceptional situations merely set an internal flag that can be tested via the `checkError` method. Optionally, a `PrintStream` can be created so as to flush automatically; this means that the `flush` method is automatically invoked after a byte array is written, one of the `println` methods is invoked, or a newline character or byte (`'\n'`) is written.

All characters printed by a `PrintStream` are converted into bytes using the platform's default character encoding. The `PrintWriter` class should be used in situations that require writing characters rather than bytes.

Since:

JDK1.0

Field Summary

Fields inherited from class java.io.FilterOutputStream

`out`

Constructor Summary

Constructors

Constructor and Description
PrintStream(**File** file) Creates a new print stream, without automatic line flushing, with the specified file.
PrintStream(**File** file, **String** csn) Creates a new print stream, without automatic line flushing, with the specified file and charset.
PrintStream(**OutputStream** out) Creates a new print stream.
PrintStream(**OutputStream** out, boolean autoFlush) Creates a new print stream.
PrintStream(**OutputStream** out, boolean autoFlush, **String** encoding) Creates a new print stream.
PrintStream(**String** fileName)

Creates a new print stream, without automatic line flushing, with the specified file name.

PrintStream(**String** fileName, **String** csn)
Creates a new print stream, without automatic line flushing, with the specified file name and charset.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
PrintStream	**append**(char c) Appends the specified character to this output stream.
PrintStream	**append**(**CharSequence** csq) Appends the specified character sequence to this output stream.
PrintStream	**append**(**CharSequence** csq, int start, int end) Appends a subsequence of the specified character sequence to this output stream.
boolean	**checkError**() Flushes the stream and checks its error state.
protected void	**clearError**() Clears the internal error state of this stream.
void	**close**() Closes the stream.
void	**flush**() Flushes the stream.
PrintStream	**format**(**Locale** l, **String** format, **Object**... args) Writes a formatted string to this output stream using the specified format string and arguments.
PrintStream	**format**(**String** format, **Object**... args) Writes a formatted string to this output stream using the specified format string and arguments.
void	**print**(boolean b) Prints a boolean value.
void	**print**(char c) Prints a character.
void	**print**(char[] s) Prints an array of characters.
void	**print**(double d) Prints a double-precision floating-point number.
void	**print**(float f) Prints a floating-point number.
void	**print**(int i) Prints an integer.
void	**print**(long l) Prints a long integer.
void	**print**(**Object** obj) Prints an object.
void	**print**(**String** s) Prints a string.
PrintStream	**printf**(**Locale** l, **String** format, **Object**... args) A convenience method to write a formatted string to this output stream using the specified format string and arguments.
PrintStream	**printf**(**String** format, **Object**... args) A convenience method to write a formatted string to this output stream using the specified format string and arguments.
void	**println**() Terminates the current line by writing the line separator string.
void	**println**(boolean x) Prints a boolean and then terminate the line.
void	**println**(char x) Prints a character and then terminate the line.
void	**println**(char[] x)

	Prints an array of characters and then terminate the line.
void	println(double x) Prints a double and then terminate the line.
void	println(float x) Prints a float and then terminate the line.
void	println(int x) Prints an integer and then terminate the line.
void	println(long x) Prints a long and then terminate the line.
void	println(Object x) Prints an Object and then terminate the line.
void	println(String x) Prints a String and then terminate the line.
protected void	setError() Sets the error state of the stream to true.
void	write(byte[] buf, int off, int len) Writes len bytes from the specified byte array starting at offset off to this stream.
void	write(int b) Writes the specified byte to this stream.

Vedlegg 11 Klassen BufferedReader

java.io

Class BufferedReader

java.lang.Object
  java.io.Reader
    java.io.BufferedReader

All Implemented Interfaces:
Closeable, AutoCloseable, Readable

Direct Known Subclasses:
LineNumberReader

---

public class **BufferedReader**
extends Reader

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.
In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

```
BufferedReader in
  = new BufferedReader(new FileReader("foo.in"));
```

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.
Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.

Since:
JDK1.1

See Also:
FileReader, InputStreamReader, Files.newBufferedReader(java.nio.file.Path, java.nio.charset.Charset)

Field Summary

Fields inherited from class java.io.Reader
lock

Constructor Summary

Constructors

Constructor and Description

BufferedReader(Reader in)
Creates a buffering character-input stream that uses a default-sized input buffer.

BufferedReader(Reader in, int sz)
Creates a buffering character-input stream that uses an input buffer of the specified size.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
void	close() Closes the stream and releases any system resources associated with it.
Stream<String>	lines() Returns a Stream, the elements of which are lines read from this BufferedReader.
void	mark(int readAheadLimit) Marks the present position in the stream.
boolean	markSupported() Tells whether this stream supports the mark() operation, which it does.
int	read() Reads a single character.
int	read(char[] cbuf, int off, int len) Reads characters into a portion of an array.
String	readLine() Reads a line of text.
boolean	ready() Tells whether this stream is ready to be read.
void	reset() Resets the stream to the most recent mark.
long	skip(long n) Skips characters.

Vedlegg 12 Klassen FileReader

java.io

Class FileReader

java.lang.Object
  java.io.Reader
    java.io.InputStreamReader

java.io.FileReader

All Implemented Interfaces:

Closeable, AutoCloseable, Readable

---

public class **FileReader**
extends InputStreamReader

Convenience class for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.

FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.

Since:

JDK1.1

See Also:

InputStreamReader, FileInputStream

Field Summary

Fields inherited from class java.io.Reader

lock

Constructor Summary

Constructors

Constructor and Description
FileReader(File file) Creates a new FileReader, given the File to read from.
FileReader(FileDescriptor fd) Creates a new FileReader, given the FileDescriptor to read from.
FileReader(String fileName) Creates a new FileReader, given the name of the file to read from.

Method Summary

Methods inherited from class java.io.InputStreamReader

close, getEncoding, read, read, ready

Methods inherited from class java.io.Reader

mark, markSupported, read, read, reset, skip

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Vedlegg 13 Klassen FileWriter

java.io
Class FileWriter
java.lang.Object

java.io.Writer

java.io.OutputStreamWriter

java.io.FileWriter

All Implemented Interfaces:
Closeable, Flushable, Appendable, AutoCloseable

---

public class **FileWriter**
extends OutputStreamWriter
Convenience class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one FileWriter (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.
FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.

Since:
JDK1.1

See Also:
OutputStreamWriter, FileOutputStream

Field Summary

Fields inherited from class java.io.Writer

lock

Constructor Summary

Constructors

Constructor and Description

FileWriter(**File** file)
Constructs a FileWriter object given a File object.

FileWriter(**File** file, boolean append)
Constructs a FileWriter object given a File object.

FileWriter(**FileDescriptor** fd)
Constructs a FileWriter object associated with a file descriptor.

FileWriter(**String** fileName)
Constructs a FileWriter object given a file name.

FileWriter(**String** fileName, boolean append)
Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

Method Summary

Methods inherited from class java.io.OutputStreamWriter

close, flush, getEncoding, write, write, write

Methods inherited from class java.io.Writer

append, append, append, write, write

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait