Referensbilagor tentamen 2019-06-03

Programmeringsteknik II 1TD722, 1TD726

1. (Detta) försättsblad
2. Kortreferens viktiga klasser
3. BST.java
4. CourseRecord.java
5. ExamException.java
6. Instruction.java
7. Misc.java
8. Node.java
9. SmallNavigator.java
10. Dokumentation ArrayList (utdrag)
11. Dokumentation Comparable
12. Dokumentation FileWriter
13. Dokumentation PrintWriter (utdrag)
14. Dokumentation Scanner (utdrag)
15. Dokumentation String (utdrag)
### Klassen Integer

**Konstruktorer**

- `Integer(String s)`
- `Integer(int x)`

**Metoder**

- `int (valueOf())`
- `static int parseInt(String s)`

### Klassen Math

**Konstruktorer och metoder**

- `static double PI`
- `static double E`
- `static double exp(double d)`
- `static double log(double d)`
- `static double sinh(double d)`
- `static double cosh(double d)`
- `static long random()`
- `static long abs(int i)`
- `static double abs(double d)`
- `static long max(long i, long j)`
- `static long min(long i, long j)`
- `static long toIntExact(int x)`
- `static long toIntExact(long x)`

### Klassen Arrays

**Konstruktor**

- `Arrays.<Integer> newX = new Arrays.<Integer>[]();`

**Metoder**

- `int size()`
- `boolean equals(Object o)`
- `boolean addAll(int index, E element)`
- `boolean remove(int index)`
- `boolean remove(Object o)`
- `Object set(int index, E element)`
- `void clear()`

### Klassen Scanner

**Konstruktor**

- `Scanner(String s)` // scan from standard input

**Metoder**

- `boolean hasNext()`
- `boolean hasNextInt()`
- `String next()`
- `int nextInt()`
- `double nextDouble()`
- `String nextLine()`
- `void close()`

### Klassen Double

**Konstruktor**

- `Double(String s)`
- `Double(double d)`

**Metoder**

- `double doubleValue()`
- `static double parseDouble(String s)`

### Klassen Exception

**Klassen RuntimeException**

**Konstruktor**

- `RuntimeException(String msg)`

**Metoder**

- `String getMessage()`

### Klassen TreeSet

**Konstruktor**

- `TreeSet<String>()`

**Metoder**

- `int size()`
- `boolean addAll(int index, E element)`
- `void remove(int index)`
- `void remove(Object o)`
- `void clear()`

### Klassen TreeMap

**Konstruktor**

- `TreeMap<K, V>()`

**Metoder**

- `V put(K k, V v)`
- `void remove(K key)`
- `V get(Object key)`
- `void clear()`
```java
import java.util.ArrayList;

/**
 * Small class implementing a binary search tree.
 */
public class BST {
  /**
   * First node in list. May be null.
   */
  private Node root;

  /**
   * Create a new tree.
   *
   * @param root Root of the new tree (not copied).
   */
  public BST(Node root) {
    this.root = root;
  }

  /**
   * Create a new tree with specified root.
   *
   * @param newn Node to add.
   * @throws ExamException If the value already exists in the tree.
   */
  public BST(Node root, Node newn) throws ExamException {
    if (newn == null) {
      return n;
    }
    // Nothing to add, just skip everything.
    if (newn == null) {
      return n;
    }
    int val = newn.data;
    if (n.data > val) {
      n.left = add(n.left, newn);
    } else if (n.data < val) {
      n.right = add(n.right, newn);
    } else {
      throw new ExamException("Value to add already present in tree.");
    }
    return n;
  }

  /**
   * @return The root.
   */
  public static Node remove(Node n, int val) throws ExamException {
    if (n == null) {
      throw new ExamException("Value to remove not present in tree.");
    }
    if (n.data == val) {
      if (n.left == null) return n.right;
      if (n.right == null) return n.left;
      // Both are non-null. We build on the right subtree and add the
      // whole left subtree at the appropriate place.
      add(n.right, n.left);
      return n.right;
    }
    if (n.data > val) {
      n.left = remove(n.left, val);
    } else if (n.data > val) {
      n.right = remove(n.right, val);
    } else {
      throw new RuntimeException("Unexpected mismatch in tree structure." +
                          "Should never happen.");
    }
    return n;
  }

  /**
   * Adds a value to the tree.
   *
   * @param val Value to add.
   * @throws ExamException If the value already exists in the tree.
   */
  public void add(int val) {
    root = add(root, val);
  }

  /**
   * @return the (possibly new) child node at this level in the tree
   */
  private static Node add(Node n, Node newn) {
    // Empty leaf location found
    if (n == null) {
      return newn;
    }
    // Nothing to add, just skip everything.
    if (newn == null) {
      return n;
    }
    int val = newn.data;
    if (n.data > val) {
      n.left = add(n.left, newn);
    } else if (n.data < val) {
      n.right = add(n.right, newn);
    } else {
      throw new ExamException("Value to add already present in tree.");
    }
    return n;
  }

  /**
   * Removes a value from the tree.
   *
   * @throws ExamException If the value is not present in the tree.
   */
  public static boolean removeIfMissing(int val) {
    // EXAM: A9
  }

  /**
   * Returns the root of the tree, to other classes in this package.
   * @return the root.
   */
  public Node getRoot() {
    return root;
  }

  /**
   * Creates a tree with n nodes, with values like 0, -1, 2, -3, 4, -5 etc.
   *
   * @param n   Element for adding.
   * @param newn Node to add.
   * @return the (possibly new) child node at this level in the tree
   */
  public static BST createPosNegTree(int n) {
    BST bst = new BST();
    for (int i = 0; i < n; i++) {
      int val = i;
      if (i % 2 == 1) {
        val *= -1;
      }
      bst.add(val);
    }
    return bst;
  }

  /*
   * Adds a value to the tree.
   *
   * @param val Value to add.
   * @return True if the value was added, false if it already was present.
   */
  public boolean addIfMissing(int val) {
    // EXAM: A9
  }
}
```

public boolean contains(int val) {
    return contains(root, val);
}

/**
 * Helper method.
 * Determines whether the value val is present in the tree.
 * @param val The value to look for.
 * @param n   The current tree node to consider.
 * @param val The value to look for.
 * @return True if and only if the element is found.
 */
private static boolean contains(Node n, int val) {
    if (n.data == val) {
        return true;
    }
    if (n.data < val) {
        return contains(n.right, val);
    } else {
        return contains(n.left, val);
    }
}

/**
 * Compute the inner path length for the tree, by summing the contribution
 * for all elements.
 */
public int innerPathLength() {
    return innerPathLength(root);
}

private static int innerPathLength(Node n) {
    if (n == null) {
        return 0;
    }
    return size(n) + innerPathLength(n.left) + innerPathLength(n.right);
}

/**
 * Reports the size of the full tree.
 */
public int size() {
    return size(root);
}

/**
 * Counts the size of the tree rooted in n.
 * @return The size of the tree, 0 if n is null.
 */
private static int size(Node n) {
    if (n == null) {
        return 0;
    }
    return 1 + size(n.left) + size(n.right);
}

/**
 * Return the largest element. Throw if empty.
 * @throws ExamException If the tree is currently empty.
 */
public int largest() {
    if (root == null) {
        throw new ExamException("Empty tree, no largest element."");
    }
    return largest(root);
}

private static int largest(Node n) {
    if (n.left == null) {
        return n.data;
    }
    return largest(n.right);
}

/**
 * Treats a clone of the full tree.
 */
@Override
public BST clone() {
    return new BST(clone(root));
}

private static Node clone(Node n) {
    if (n == null) {
        return null;
    }
    return new Node(n.data,
        clone(n.left),
        clone(n.right));
}

/**
 * Prints all the data in the tree in ascending order, one value per line.
 * @param minSum Minimum sum to achieve.
 * @param maxSum Maximum sum to achieve.
 * @return The size of the tree, 0 if n is null.
 */
public int minCountSummingTo(int minSum) {
    BST copy = clone();
    int sum = 0;
    int count = 0;
    for (; sum < minSum; count++) {
        int val = copy.largest();
        copy.remove(val);
        sum += val;
    }
    return count;
}
```java
import java.util.ArrayList;
import java.util.List;

public class CourseRecord implements Comparable<CourseRecord> {
    /**
     * Name of course.
     */
    private final String course;
    /**
     * Name of student.
     */
    private final String student;
    /**
     * Grade of student in course, or zero to indicate no grade set yet.
     * Valid set grade values are in the set {3,4,5}.
     * A grade cannot go from being set to being unset. A set grade
     * cannot be lowered or unset, but possibly increased ("plussa").
     */
    private int grade = 0;

    public CourseRecord(String course, String student) {
        this(course, student, 0);
    }

    public CourseRecord(String course, String student, int grade) {
        // Call the three-param constructor with 0 for implicit grade.
        this(course, student, 0);
    }

    public CourseRecord(String course, String student) {
        // EXAM: A4
        this(course, student, 0);
    }

    public CourseRecord(String course) {
        // EXAM: A4
        this(course, 0);
    }

    /**
     * Get the name of the course.
     */
    public String getCourse() {
        return course;
    }

    /**
     * Get the name of the student.
     */
    public String getStudent() {
        return student;
    }

    /**
     * Get the grade (0 indicates grade not actually set yet).
     * @return The grade.
     */
    public int getGrade() {
        // return grade;
    }

    /**
     * Set the grade. 0 indicates grade not set.
     * 3,4,5 are valid values. Set grades can only be increased.
     * @param grade A valid grade value. Only valid values are 0, 3, 4, 5.
     * @throws ExamException For any empty or null string, or for invalid grade values.
     */
    public void setGrade(int grade) {
        // EXAM: A4
        if (grade == g.grade && course.equals(g.course) &&
                student.equals(g.student));
    }

    /**
     * Returns a hash code representing this course record. Different objects
     * should, if possible, have different hash codes. Equivalent objects
     * need to be assigned identical hash codes.
     */
    @Override
    public int hashCode() {
        // EXAM: A2
        return 0;
    }

    /**
     * Compare this object to another object (presumably a CourseRecord).
     * @param o The other object.
     * @return Whether they are identical
     * @throws ExamException For any empty or null string.
     */
    @Override
    public boolean equals(Object o) {
        if (o == null || o.getClass() != getClass()) {
            return false;
        }
        CourseRecord g = (CourseRecord) o;
        return grade == g.grade && course.equals(g.course) &&
                student.equals(g.student);
    }

    /**
     * Compare this CourseRecord to another CourseRecord, sorting first by
     * course (ascending), then by grade (descending), then by student name
     * (ascending).
     * @param o Other GradeRcord.
     * @return Negative value if this comes before o, 0 if equal, positive
     *         if this comes after o.
     * @throws ExamException For any empty or null string.
     */
    @Override
    public int compareTo(Object o) {
        if (o == null || o.getClass() != getClass()) {
            return false;
        }
        CourseRecord g = (CourseRecord) o;
        return grade == g.grade && course.equals(g.course) &&
                student.equals(g.student);
    }

    public static ArrayList<CourseRecord> getRecordsForCourse(String course, List<CourseRecord> list) {
        ArrayList<CourseRecord> res = new ArrayList<CourseRecord>();
        for (CourseRecord cr : list) {
            if (cr.getCourse().equals(course)) {
                res.add(cr);
            }
        }
        return res;
    }
}
```
    return res;
};

/**
 * Compares this course record to another, returning true if this one is
 * strictly better, that is, if they apply to the same course (but not
 * necessarily the same student), both have grades set and the grade for
 * this record is higher. A course record cannot be strictly better than
 * a null reference.
 *
 * @param other The possibly null object to compare against.
 *
 * @return True if the two course records apply to the same course, there
 * is a grade recorded for both, and the grade for this one is strictly
 * larger.
 */
public boolean strictlyBetterThan(CourseRecord other) {
    // EXAM: A8
    
    /**
     * Represents the course name, student name, the presence of a recorded
     * grade, and if so the value of that grade, in a human-friendly manner
     * where the different entries are clearly separated and distinguishable.
     */
    // EXAM: A10
```java
/**
 * A class for most exceptional results during the exam.
 */
public class ExamException extends RuntimeException {

    /**
     * Create the exception, using the specified message.
     * @param msg Message string.
     */
    public ExamException(String msg) {
        super(msg);
    }
}
```
import java.io.IOException;
import java.util.ArrayList;

/**
 * Base class with main method and helper methods for stack machine.
 *
 * @param stack The current stack.
 * @param ip The current instruction pointer.
 * @return The next instruction pointer to use, depending on the
 * instance variables.
 */
abstract class Instruction {

    /**
     * Helper method to pop an element from the stack.
     *
     * @return The popped value.
     */
    @Override
    public int execute(ArrayList<Integer> stack, int ip) {
        return ip + deltaEqual;
    }

    /**
     * Helper method to peek at the top element on the stack.
     *
     * @param stack The stack.
     */
    protected static int peek(ArrayList<Integer> stack) {
        return stack.get(stack.size() - 1);
    }

    /**
     * Helper method to push an element to the stack.
     *
     * @param stack The stack.
     */
    protected static void push(ArrayList<Integer> stack, int val) {
        stack.add(val);
    }

    /**
     * An equality test between a constant and the top value on the stack.
     */
    private static class Equals extends Instruction {
        /**
         * The value to compare against.
         */
        private int value;

        /**
         * The number of instructions to jump if we have equality.
         */
        private int deltaEqual;

        /**
         * The number of instructions to jump if we do not have equality.
         */
        private int deltaNotEqual;

        /** Constructor setting fields. */
        public Equals(int value, int deltaEqual, int deltaNotEqual) {
            this.value = value;
            this.deltaEqual = deltaEqual;
            this.deltaNotEqual = deltaNotEqual;
        }

        /** Execute the equality test.
         *
         * @param stack The current stack.
         * @param ip The current instruction pointer.
         * @return The next instruction pointer to use, depending on the
         * instance variables.
         */
        @Override
        public int execute(ArrayList<Integer> stack, int ip) {
            if (peek(stack) == value) {
                return ip + deltaEqual;
            } else {
                return ip + deltaNotEqual;
            }
        }
    }

    /**
     * Base class for all instructions that naturally step ahead one
     * instruction.
     */
    protected static class NextStepper extends Instruction {
        /**
         * Just step to the next instruction.
         *
         * @param stack The current stack.
         * @param ip The current instruction pointer.
         * @return The next instruction pointer to use, ip + 1.
         */
        @Override
        public int execute(ArrayList<Integer> stack, int ip) {
            return ip + 1;
        }
    }

    /**
     * Push a constant onto the stack.
     */
    private static class Push extends NextStepper {
        /**
         * The value to push.
         */
        private int value;

        /** Constructor setting value. */
        public Push(int value) {
            this.value = value;
        }

        /**
         * Push a constant onto the stack, and advance the instruction pointer
         * by 1.
         *
         * @param stack The current stack.
         * @param ip The current instruction pointer.
         * @return The next instruction pointer to use, ip + 1.
         */
        @Override
        public int execute(ArrayList<Integer> stack, int ip) {
            push(stack, value);
            return super.execute(stack, ip);
        }
    }

    /** Swap the two top elements on the stack. */
    private static class Swap extends NextStepper {
        /**
         * The two top values.
         */
        private int top1;
        private int top2;

        /** Constructor setting top values. */
        public Swap() {
            this.top1 = peek(stack);
            this.top2 = peek(stack);
        }

        /**
         * Swap the two top elements on the stack.
         *
         * @param stack The current stack.
         * @param ip The current instruction pointer.
         * @return The next instruction pointer to use, ip + 1.
         */
        @Override
        public int execute(ArrayList<Integer> stack, int ip) {
            return super.execute(stack, ip);
        }
    }
}
*/
private static class Swap extends NextStepper {
/**
 * Swap the two top stack elements, and advance the instruction
 * pointer.
 * @param stack The current stack.
 * @param ip The current instruction pointer.
 * @return The next instruction pointer to use, ip + 1.
 */
public int execute(ArrayList<Integer> stack, int ip) {
    int oldTop = pop(stack);
    int belowTop = pop(stack);
    push(stack, oldTop);
    push(stack, belowTop);
    return super.execute(stack, ip);
}
/**
* @param stack The current stack.
* @param ip The current instruction pointer.
* @return The next instruction pointer to use, ip + 1.
*/
@Override
private static class Copy extends NextStepper {
    /**
     * Copy the top element on the stack.
     * @param stack The current stack.
     * @param ip The current instruction pointer.
     * @return The next instruction pointer to use, ip + 1.
     */
    public int execute(ArrayList<Integer> stack, int ip) {
        int top = pop(stack);
        push(stack, top);
        return super.execute(stack, ip);
    }
    /**
     * Pops an element from the stack and prints it using an abstract print
     * method.
     * @param stack The current stack.
     * @param ip The current instruction pointer.
     * @return The next instruction pointer to use, ip + 1.
     */
    @Override
    private static class IntPrint extends Print {
        /**
         * Prints the value popped from the stack.
         * @param stack The current stack.
         * @param ip The current instruction pointer.
         * @return The next instruction pointer to use, ip + 1.
         */
         public int execute(ArrayList<Integer> stack, int ip) {
             int top = pop(stack);
             push(stack, top);
             return super.execute(stack, ip);
         }
        /**
         * @param val The value to print.
         */
         @Override
         void print(int val) {
             System.out.print(val);
         }
    }
    /**
     * Pops an element from the stack and prints it as a character.
     * @param stack The current stack.
     * @param ip The current instruction pointer.
     * @return The next instruction pointer to use, ip + 1.
     */
    @Override
    private static class CharPrint extends Print {
        /**
         * Prints the value popped from the stack.
         * @param stack The current stack.
         * @param ip The current instruction pointer.
         * @return The next instruction pointer to use, ip + 1.
         */
         public int execute(ArrayList<Integer> stack, int ip) {
             int top = pop(stack);
             push(stack, (char) top);
             return super.execute(stack, ip);
         }
        /**
         * Prints a newline.
         */
        protected void print(char val) {
            System.out.print((char) val);
        }
    }
    /**
     * Create a demo program of our stack machine.
     */
    private static ArrayList<Instruction> createProgram() {
        ArrayList<Instruction> is = new ArrayList<Instruction>();
        is.add(new Push(1));
        return is;
    }
    /**
     * Prints a newline character.
     * @param stack The current stack.
     * @param ip The current instruction pointer.
     * @return The next instruction pointer to use, ip + 1.
     */
    @Override
    private static class PrintLn extends NextStepper {
        /**
         * Prints a newline.
         */
         protected void print(char val) {
             System.out.println(val);
         }
    }
    /**
     * Pops the top two elements on the stack, and push their sum.
     * @param stack The current stack.
     * @param ip The current instruction pointer.
     * @return The next instruction pointer to use, ip + 1.
     */
    @Override
    private static class Add extends NextStepper {
        /**
         * Read top two elements on the stack, push their sum, and advance
         * the instruction pointer by 1.
         * @param stack The current stack.
         * @param ip The current instruction pointer.
         * @return The next instruction pointer to use, ip + 1.
         */
         public int execute(ArrayList<Integer> stack, int ip) {
             int a = pop(stack);
             int b = pop(stack);
             push(stack, a + b);
             return super.execute(stack, ip + 1);
         }
        /**
         * Pops an element from the stack and prints it as a number.
         * @param stack The current stack.
         * @param ip The current instruction pointer.
         * @return The next instruction pointer to use, ip + 1.
         */
        @Override
        private static class Int extends NextStepper {
            /**
             * Prints the value popped from the stack.
             * @param stack The current stack.
             * @param ip The current instruction pointer.
             * @return The next instruction pointer to use, ip + 1.
             */
             public int execute(ArrayList<Integer> stack, int ip) {
                 int top = pop(stack);
                 return super.execute(stack, ip);
             }
            /**
             * @param val The value to print.
             */
             @Override
             void print(int val) {
                 System.out.println(val);
             }
         }
    }
}
is.add(new Push(0));
is.add(new Add());
is.add(new Swap());
is.add(new Copy());
is.add(new Add());
is.add(new Push(1));
is.add(new Add());
is.add(new Swap());
is.add(new New Eq(8, 1, -6)); // Jump back to swap if !=
is.add(new Swap());
is.add(new IntPrint());
is.add(new PrintLn());
is.add(new Push(2));
is.add(new Push(2));
is.add(new Swaps());
is.add(new PrintLn());

    return is;
}

/**
 * Menu−driven looping program for entering student information.
 *
 * @throws IOException For any IO error while writing.
 */

public static void main(String[] args) throws IOException {
    ArrayList<Instruction> is = createProgram();
    ArrayList<Integer> stack = new ArrayList<Integer>();
    int ip = 0;
    while (ip >= 0 & ip < is.size()) {
        Instruction now = is.get(ip);
        ip = now.execute(stack, ip);
    }
public class Misc {

    public static boolean matchBrackets(String text) {
        // EXAM: B1
    }

    public static int[] sortNonNegInts(int[] array) {
        // EXAM: B2
    }

    public static void main(String[] args) throws IOException {
        // EXAM: B5
    }
}
/**
 * Package-internal class for BST nodes.
 */

class Node {
    /**
     * Stored value.
     */
    public int data;

    /**
     * Root node in left subtree.
     */
    public Node left;

    /**
     * Root node in right subtree.
     */
    public Node right;

    /**
     * Constructor. No argument checking!
     *
     * @param data  Data to store.
     * @param left  Left-hand node.
     * @param right Right-hand node.
     */
    public Node(int data, Node left, Node right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
import java.io.IOException;

public class SmallNavigator extends KeypressListener {
    private Node root;
    private Node now;

    public SmallNavigator(Node root) {
        this.root = root;
        now = root;
        printPrompt();
    }

    public void keypress(char c) {
        int c = System.in.read();
        while ((c = System.in.read()) != -1) {
            app.keypress((char) c);
        }
    }

    public static BST createTree() {
        BST bst = new BST();
        bst.add(5);
        bst.add(3);
        bst.add(4);
        bst.add(7);
        bst.add(9);
        bst.add(14);
        bst.add(12);
        bst.add(13);
        bst.add(16);
        bst.add(15);
        bst.add(17);
        return bst;
    }

    public static void main(String[] args) throws IOException {
        KeypressListener app = new SmallNavigator(createTree().getRoot());
        while ((c = System.in.read()) != -1) {
            app.keypress((char) c);
        }
    }
}
Class ArrayList\<E\>  

- java.lang.Object  
- java.util.AbstractCollection\<E\>  
- java.util.AbstractList\<E\>  
- java.util.ArrayList\<E\>  

All Implemented Interfaces:  
  Serializable, Cloneable, Iterable\<E\>, Collection\<E\>, List\<E\>, RandomAccess

Direct Known Subclasses:  
  AttributeList, RoleList, RoleUnresolvedList

public class ArrayList\<E\>  
extends AbstractList\<E\>  
implements List\<E\>, RandomAccess, Cloneable, Serializable

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

- The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding an elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

- Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

- An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapulates the list. If no such object exists, the list should be "wrapped" using the Collections.synchronizedList method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```java
List list = Collections.synchronizedList(new ArrayList(...));
```

The iterators returned by this class's iterator and listIterator methods are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

This class is a member of the Java Collections Framework.

Since: 1.2

See Also: Collection, List, LinkedList, Vector, Serialized Form

- Field Summary
- Constructor Summary
- Method Summary
### Klassdokumentation 3

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean add (E e)</td>
<td>Appends the specified element to the end of this list.</td>
</tr>
<tr>
<td>void add (int index, E element)</td>
<td>Inserts the specified element at the specified position in this list.</td>
</tr>
<tr>
<td>boolean addAll (Collection&lt;? extends E&gt; c)</td>
<td>Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection’s Iterator.</td>
</tr>
<tr>
<td>boolean addAll (int index, Collection&lt;? extends E&gt; c)</td>
<td>Inserts all of the elements in the specified collection into this list, starting at the specified position.</td>
</tr>
<tr>
<td>void clear ()</td>
<td>Removes all of the elements from this list.</td>
</tr>
<tr>
<td>Object clone ()</td>
<td>Returns a shallow copy of this ArrayList instance.</td>
</tr>
<tr>
<td>boolean contains (Object o)</td>
<td>Returns true if this list contains the specified element.</td>
</tr>
<tr>
<td>void ensureCapacity (int minCapacity)</td>
<td>Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.</td>
</tr>
<tr>
<td>void forEach (Consumer&lt;? super E&gt; action)</td>
<td>Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.</td>
</tr>
</tbody>
</table>

### Klassdokumentation 4

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>get (int index) Returns the element at the specified position in this list.</td>
</tr>
<tr>
<td>int</td>
<td>indexOf (Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.</td>
</tr>
<tr>
<td>boolean</td>
<td>isEmpty () Returns true if this list contains no elements.</td>
</tr>
<tr>
<td>Iterator&lt;E&gt;</td>
<td>iterator () Returns an iterator over the elements in this list in proper sequence.</td>
</tr>
<tr>
<td>Iterator&lt;E&gt;</td>
<td>lastIndexOf (Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.</td>
</tr>
<tr>
<td>ListIterator&lt;E&gt;</td>
<td>listIterator () Returns a list iterator over the elements in this list (in proper sequence).</td>
</tr>
<tr>
<td>ListIterator&lt;E&gt;</td>
<td>listIterator (int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.</td>
</tr>
<tr>
<td>E</td>
<td>remove (int index) Removes the element at the specified position in this list.</td>
</tr>
<tr>
<td>boolean</td>
<td>remove (Object o) Removes the first occurrence of the specified element from this list, if it is present.</td>
</tr>
<tr>
<td>boolean</td>
<td>removeAll (Collection&lt;?&gt; c) Removes from this list all of its elements that are contained in the specified collection.</td>
</tr>
<tr>
<td>boolean</td>
<td>removeIf (Predicate&lt;? super E&gt; filter)</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>protected void</td>
<td>removeRange(int fromIndex, int toIndex)</td>
</tr>
<tr>
<td>void</td>
<td>replaceAll (UnaryOperator&lt;E&gt; operator)</td>
</tr>
<tr>
<td>boolean</td>
<td>retainAll (Collection&lt;?&gt; c)</td>
</tr>
<tr>
<td>E</td>
<td>set(int index, E element)</td>
</tr>
<tr>
<td>int</td>
<td>size ()</td>
</tr>
<tr>
<td>void</td>
<td>sort (Comparator&lt;? super E&gt; c)</td>
</tr>
<tr>
<td>Spliterator&lt;E&gt;</td>
<td>spliterator ()</td>
</tr>
<tr>
<td>List&lt;E&gt;</td>
<td>subList(int fromIndex, int toIndex)</td>
</tr>
<tr>
<td>Object []</td>
<td>toArray ()</td>
</tr>
<tr>
<td>&lt;T&gt; T[]</td>
<td>toArray (T[] a)</td>
</tr>
<tr>
<td>void</td>
<td>trimToSize ()</td>
</tr>
</tbody>
</table>

- Methods inherited from class java.util.AbstractList
equals, hashCode

- Methods inherited from class java.util.AbstractCollection
containsAll, toString

- Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

- Methods inherited from interface java.util.List
containsAll, equals, hashCode

- Methods inherited from interface java.util.Collection

parallelStream, stream
public interface Comparable<T>

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class’s natural ordering, and the class’s `compareTo` method is referred to as its natural comparison method.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort((Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

The natural ordering for a class C is said to be consistent with equals if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class C. Note that null is not an instance of any class, and `e1.compareTo(null)` should throw a NullPointerException even though `e1.equals(null)` returns false.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave “strangely” when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.

For example, if one adds two keys `a` and `b` such that `(a.equals(b) && a.compareTo(b) == 0)` to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because `a` and `b` are equivalent from the sorted set’s perspective.

Virtually all Java core classes that implement `Comparable` have natural orderings that are consistent with equals. One exception is `java.math.BigDecimal`, whose natural ordering equates `BigDecimal` objects with equal values and different precisions (such as 4.0 and 4.00),...
It is strongly recommended, but not strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(expression)` designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.

### Parameters:
- `o` - the object to be compared.

### Returns:
- a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

### Throws:
- `NullPointerException` - if the specified object is null
- `ClassCastException` - if the specified object’s type prevents it from being compared to this object.

---

### Field Summary

#### Fields inherited from class `java.io.Writer`
- `lock`

### Constructor Summary

#### Constructors

- `FileWriter(File file)`
  Constructs a `FileWriter` object given a `File` object.

- `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.

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

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

### 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`
Constructor Detail

**FileWriter**

*public* `FileWriter(String fileName)`

**throws** IOException

Constructs a FileWriter object given a file name.

**Parameters:**

- `fileName` - String The system-dependent filename.

**Throws:**

- IOException - if the named file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

*public* `FileWriter(String fileName, boolean append)`

**throws** IOException

Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

**Parameters:**

- `fileName` - String The system-dependent filename.
- `append` - boolean if true, then data will be written to the end of the file rather than the beginning.

**Throws:**

- IOException - if the named file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

*public* `FileWriter(File file)`

**throws** IOException

Constructs a FileWriter object given a File object.

**Parameters:**

- `file` - a File object to write to.

**Throws:**

- IOException - if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

*public* `FileWriter(File file, boolean append)`

**throws** IOException

Constructs a FileWriter object given a File object.

**Parameters:**

- `file` - a File object to write to.
- `append` - boolean if true, then data will be written to the end of the file rather than the beginning.

**Throws:**

- IOException - if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason

**Since:** 1.4

Class PrintWriter

*public* `PrintWriter(FileDescriptor fd)`

Constructs a PrintWriter object associated with a file descriptor.

**Parameters:**

- `fd` - FileDescriptor object to write to.

**java.io**

**All Implemented Interfaces:**

- Closeable
- Flushable
- Appendable
- AutoCloseable

**public class PrintWriter**

*extends* Writer

Prints formatted representations of objects to a text-output stream. This class implements all of the `print` methods found in PrintStream. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

Unlike the PrintStream class, if automatic flushing is enabled it will be done only when one of the `println`, `printf`, or `format` methods is invoked, rather than whenever a newline character happens to be output. These methods use the platform's own notion of line separator rather than the newline character.

Methods in this class never throw I/O exceptions, although some of its constructors may. The client may inquire as to whether any errors have occurred by invoking `checkError()`.
Since: JDK1.1

### Field Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Field and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected Writer</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>The underlying character-output stream of this PrintWriter.</td>
</tr>
</tbody>
</table>

Fields inherited from class java.io.Writer

- lock

### Constructor Summary

<table>
<thead>
<tr>
<th>Constructor and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrintWriter (File file)</td>
</tr>
<tr>
<td>Creates a new PrintWriter, without automatic line flushing, with the specified file.</td>
</tr>
</tbody>
</table>

**PrintWriter (File file, String csn)**

Creates a new PrintWriter, without automatic line flushing, with the specified file and charset.

**PrintWriter (OutputStream out)**

Creates a new PrintWriter, without automatic line flushing, from an existing OutputStream.

**PrintWriter (OutputStream out, boolean autoFlush)**

Creates a new PrintWriter from an existing OutputStream.

**PrintWriter (String fileName)**

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

**PrintWriter (String fileName, String csn)**

Creates a new PrintWriter, without automatic line flushing, with the specified file name and charset.

### Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrintWriter</td>
<td>append (char c)</td>
</tr>
<tr>
<td></td>
<td>Appends the specified character to this writer.</td>
</tr>
</tbody>
</table>

**PrintWriter | append (CharSequence csq)**

Appends the specified character sequence to this writer.

**PrintWriter | append (CharSequence csq, int start, int end)**

Appends a subsequence of the specified character sequence to this writer.

**boolean | checkError ()**

Flushes the stream if it's not closed and checks its error state.

**protected void | clearError ()**

Clears the error state of this stream.

**void | close ()**

Closes the stream and releases any system resources associated with it.

**void | flush ()**

Flushes the stream.

**PrintWriter | format (Locale l, String format, Object... args)**

Writes a formatted string to this writer using the specified format string and arguments.

**PrintWriter | format (String format, Object... args)**

Writes a formatted string to this writer 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.

**PrintWriter** **printf**(Locale l, String format, Object... args)
A convenience method to write a formatted string to this writer using the specified format string and arguments.

**PrintWriter** **printf**(String format, Object... args)
A convenience method to write a formatted string to this writer 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 value and then terminates the line.

void **println**(char x)
Prints a character and then terminates the line.

void **println**(char[] x)
Prints an array of characters and then terminates the line.

void **println**(double x)
Prints a double-precision floating-point number and then terminates the line.

void **println**(float x)
Prints a floating-point number and then terminates the line.

void **println**(int x)
Prints an integer and then terminates the line.

void **println**(long x)
Prints a long integer and then terminates the line.

void **println**(Object x)
Prints an Object and then terminates the line.

void **println**(String x)
Prints a String and then terminates the line.

protected void **setError**()
Indicates that an error has occurred.

void **write**(char[] buf)
Writes an array of characters.

void **write**(char[] buf, int off, int len)
Writes A Portion of an array of characters.

void **write**(int c)
Writes a single character.

void **write**(String s)
void write(String s, int off, int len)

Writes a portion of a string.

Methods inherited from class java.lang.Object

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

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);
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 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.

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.
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.

### Constructor Summary

<table>
<thead>
<tr>
<th>Constructor and Description</th>
<th>Constructor and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor and Description</td>
<td>Constructor and Description</td>
</tr>
<tr>
<td><code>Scanner(File source)</code></td>
<td><code>Scanner(InputStream source)</code></td>
</tr>
<tr>
<td><code>Scanner(File source, String charsetName)</code></td>
<td><code>Scanner(InputStream source, String charsetName)</code></td>
</tr>
<tr>
<td><code>Scanner(Readable source)</code></td>
<td><code>Scanner(ReadableByteChannel source)</code></td>
</tr>
<tr>
<td><code>Scanner(ReadableByteChannel source, String charsetName)</code></td>
<td><code>Scanner(String source)</code></td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void</code></td>
<td><code>close()</code></td>
</tr>
<tr>
<td></td>
<td>Closes this scanner.</td>
</tr>
<tr>
<td><code>Pattern</code></td>
<td><code>delimiter()</code></td>
</tr>
<tr>
<td></td>
<td>Returns the Pattern this Scanner is currently using to match delimiters.</td>
</tr>
<tr>
<td><code>String</code></td>
<td><code>findInRange(Pattern pattern)</code></td>
</tr>
<tr>
<td></td>
<td>Attempts to find the next occurrence of the specified pattern ignoring delimiters.</td>
</tr>
<tr>
<td><code>String</code></td>
<td><code>findInRange(String pattern)</code></td>
</tr>
<tr>
<td></td>
<td>Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.</td>
</tr>
<tr>
<td><code>String</code></td>
<td><code>findWithinHorizon(Pattern pattern, int horizon)</code></td>
</tr>
<tr>
<td></td>
<td>Attempts to find the next occurrence of the specified pattern.</td>
</tr>
<tr>
<td><code>String</code></td>
<td><code>findWithinHorizon(String pattern, int horizon)</code></td>
</tr>
<tr>
<td></td>
<td>Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.</td>
</tr>
<tr>
<td><code>boolean</code></td>
<td><code>hasNext()</code></td>
</tr>
<tr>
<td></td>
<td>Returns true if this scanner has another token in its input.</td>
</tr>
<tr>
<td><code>boolean</code></td>
<td><code>hasNext(Pattern pattern)</code></td>
</tr>
<tr>
<td></td>
<td>Returns true if the next complete token matches the specified pattern.</td>
</tr>
<tr>
<td><code>boolean</code></td>
<td><code>hasNext(String pattern)</code></td>
</tr>
<tr>
<td></td>
<td>Returns true if the next token matches the pattern constructed from the specified string.</td>
</tr>
<tr>
<td><code>boolean</code></td>
<td><code>hasNextBigDecimal()</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```java
void close()
Closes this scanner.

Pattern delimiter()
Returns the Pattern this Scanner is currently using to match delimiters.

String findInRange(Pattern pattern)
Attempts to find the next occurrence of the specified pattern ignoring delimiters.

String findInRange(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()
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNextBigDecimal()</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a BigDecimal using the <code>nextBigDecimal()</code> method.</td>
</tr>
<tr>
<td>hasNextBigInteger()</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a BigInteger in the default radix using the <code>nextBigInteger()</code> method.</td>
</tr>
<tr>
<td>hasNextBigInteger(int radix)</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a BigInteger in the specified radix using the <code>nextBigInteger()</code> method.</td>
</tr>
<tr>
<td>hasNextBoolean()</td>
<td>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 &quot;true</td>
</tr>
<tr>
<td>hasNextByte()</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a byte value in the default radix using the <code>nextByte()</code> method.</td>
</tr>
<tr>
<td>hasNextByte(int radix)</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a byte value in the specified radix using the <code>nextByte()</code> method.</td>
</tr>
<tr>
<td>hasNextDouble()</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a double value using the <code>nextDouble()</code> method.</td>
</tr>
<tr>
<td>hasNextFloat()</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a float value using the <code>nextFloat()</code> method.</td>
</tr>
<tr>
<td>hasNextInt()</td>
<td>Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the <code>nextInt()</code> method.</td>
</tr>
<tr>
<td>hasNextInt(int radix)</td>
<td>Returns true if the next token in this scanner's input can be interpreted as an int value in the specified radix using the <code>nextInt()</code> method.</td>
</tr>
<tr>
<td>hasNextLine()</td>
<td>Returns true if there is another line in the input of this scanner.</td>
</tr>
<tr>
<td>hasNextLong()</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the <code>nextLong()</code> method.</td>
</tr>
<tr>
<td>hasNextLong(int radix)</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a long value in the specified radix using the <code>nextLong()</code> method.</td>
</tr>
<tr>
<td>hasNextShort()</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a short value in the default radix using the <code>nextShort()</code> method.</td>
</tr>
<tr>
<td>hasNextShort(int radix)</td>
<td>Returns true if the next token in this scanner's input can be interpreted as a short value in the specified radix using the <code>nextShort()</code> method.</td>
</tr>
<tr>
<td>ioException()</td>
<td>Returns the <code>IOException</code> last thrown by this Scanner's underlying <code>Readable</code>.</td>
</tr>
<tr>
<td>locale()</td>
<td>Returns this scanner's locale.</td>
</tr>
<tr>
<td>match()</td>
<td>Returns the match result of the last scanning operation performed by this scanner.</td>
</tr>
<tr>
<td>next(Pattern pattern)</td>
<td>Finds and returns the next complete token from this scanner.</td>
</tr>
<tr>
<td>next(String pattern)</td>
<td>Returns the next token if it matches the specified pattern.</td>
</tr>
<tr>
<td>Class</td>
<td>Method</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>BigDecimal</td>
<td><code>nextBigDecimal()</code></td>
</tr>
<tr>
<td>BigInteger</td>
<td><code>nextBigInteger()</code></td>
</tr>
<tr>
<td>BigInteger</td>
<td><code>nextBigInteger(int radix)</code></td>
</tr>
<tr>
<td>boolean</td>
<td><code>nextBoolean()</code></td>
</tr>
<tr>
<td>byte</td>
<td><code>nextByte()</code></td>
</tr>
<tr>
<td>byte</td>
<td><code>nextByte(int radix)</code></td>
</tr>
<tr>
<td>double</td>
<td><code>nextDouble()</code></td>
</tr>
<tr>
<td>float</td>
<td><code>nextFloat()</code></td>
</tr>
<tr>
<td>int</td>
<td><code>nextInt()</code></td>
</tr>
<tr>
<td>int</td>
<td><code>nextInt(int radix)</code></td>
</tr>
<tr>
<td>String</td>
<td><code>nextLine()</code></td>
</tr>
<tr>
<td>long</td>
<td><code>nextLong()</code></td>
</tr>
<tr>
<td>long</td>
<td><code>nextLong(int radix)</code></td>
</tr>
<tr>
<td>short</td>
<td><code>nextShort()</code></td>
</tr>
<tr>
<td>short</td>
<td><code>nextShort(int radix)</code></td>
</tr>
<tr>
<td>int</td>
<td><code>radix()</code></td>
</tr>
<tr>
<td>void</td>
<td><code>remove()</code></td>
</tr>
<tr>
<td>Scanner</td>
<td><code>reset()</code></td>
</tr>
<tr>
<td>Scanner</td>
<td><code>skip(Pattern pattern)</code></td>
</tr>
<tr>
<td>Scanner</td>
<td><code>skip(String pattern)</code></td>
</tr>
<tr>
<td>String</td>
<td><code>toString()</code></td>
</tr>
<tr>
<td>Scanner</td>
<td><code>useDelimiter(Pattern pattern)</code></td>
</tr>
<tr>
<td>Scanner</td>
<td><code>useDelimiter(String pattern)</code></td>
</tr>
<tr>
<td>Scanner</td>
<td><code>useLocale(Locale locale)</code></td>
</tr>
<tr>
<td>Scanner</td>
<td><code>useRadix(int radix)</code></td>
</tr>
</tbody>
</table>
Sets this scanner’s default radix to the specified radix.

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

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

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

### Constructor Detail

**Scanner**

```java
public Scanner(InputStream source)
```

Constructs a new Scanner that produces values scanned from the specified input stream. Bytes from the stream are converted into characters using the underlying platform's default charset.

**Parameters:**

- `source` - An input stream to be scanned

---

**Scanner**

```java
public Scanner(String source)
```

Constructs a new Scanner that produces values scanned from the specified string.

**Parameters:**

- `source` - A string to scan

---

**nextLine**

```java
public String nextLine()
```

Advances this scanner past the current line and returns the input that was skipped. This method returns the rest of the current line, excluding any line separator at the end. The position is set to the beginning of the next line.

**Since this method continues to search through the input looking for a line separator, it may buffer all of the input searching for the line to skip if no line separators are present.**

**Returns:**

- the line that was skipped

**Throws:**

- `NoSuchElementException` - if no line was found

---

**nextInt**

```java
public int nextInt()
```

Scans the next token of the input as an int.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

**Returns:**

- the int scanned from the input

**Throws:**

- `InputMismatchException` - if the next token does not match the `Integer` regular expression, or is out of range
- `NoSuchElementException` - if input is exhausted
- `IllegalStateException` - if this scanner is closed
The `String` class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```java
String str = "abc";
```

is equivalent to:

```java
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Here are some more examples of how strings can be used:

```java
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

The class `String` includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the `Character` class.

The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. String concatenation is implemented through the `StringBuilder` (or `StringBuffer`) class and its `append` method. String conversions are implemented through the method `toString`, defined by `Object` and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, *The Java Language Specification*.

Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a `NullPointerException` to be thrown.
This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.

```java
String (byte[] bytes, int offset, int length)
Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.
```

```java
String (byte[] bytes, int offset, int length, Charset charset)
Constructs a new `String` by decoding the specified subarray of bytes using the specified `charset`.  
```

```java
String (byte[] ascii, int hibyte, int offset, int count)
Deprecated.
This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.
```

```java
String (byte[] bytes, int offset, int length, String charsetName)
Constructs a new `String` by decoding the specified subarray of bytes using the specified `charset`.
```

```java
String (char[] value)
Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument.
```

```java
String (char[] value, int offset, int count)
Allocates a new `String` that contains characters from a subarray of the character array argument.
```

```java
String (int[] codePoints, int offset, int count)
Allocates a new `String` that contains characters from a subarray of the Unicode code point array argument.
```

```java
String (String original)
Initializes a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
```

```java
String (StringBuffer buffer)
Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.
```

```java
String (StringBuilder builder)
```

Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

## Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java</code></td>
<td><code>char</code> <code>charAt</code> (int index)</td>
</tr>
<tr>
<td></td>
<td>Returns the char value at the specified index.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>int</code> <code>compareTo</code> (String anotherString)</td>
</tr>
<tr>
<td></td>
<td>Compares two strings lexicographically.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>int</code> <code>compareToIgnoreCase</code> (String str)</td>
</tr>
<tr>
<td></td>
<td>Compares two strings lexicographically, ignoring case differences.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>int</code> <code>codePointAt</code> (int index)</td>
</tr>
<tr>
<td></td>
<td>Returns the character (Unicode code point) at the specified index.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>int</code> <code>codePointBefore</code> (int index)</td>
</tr>
<tr>
<td></td>
<td>Returns the character (Unicode code point) before the specified index.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>int</code> <code>codePointCount</code> (int beginIndex, int endIndex)</td>
</tr>
<tr>
<td></td>
<td>Returns the number of Unicode code points in the specified text range of this <code>String</code>.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>String</code> <code>concat</code> (String str)</td>
</tr>
<tr>
<td></td>
<td>Concatenates the specified string to the end of this <code>String</code>.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>boolean</code> <code>contains</code> (CharSequence s)</td>
</tr>
<tr>
<td></td>
<td>Returns true if and only if this <code>String</code> contains the specified sequence of char values.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>boolean</code> <code>contentEquals</code> (CharSequence cs)</td>
</tr>
<tr>
<td></td>
<td>Compares this <code>String</code> to the specified <code>CharSequence</code>.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>boolean</code> <code>contentEquals</code> (StringBuffer sb)</td>
</tr>
<tr>
<td></td>
<td>Compares this <code>String</code> to the specified <code>StringBuffer</code>.</td>
</tr>
<tr>
<td><code>java</code></td>
<td><code>String</code> <code>copyValueOf</code> (char[] data)</td>
</tr>
</tbody>
</table>
static String copyValueOf(char[] data, int offset, int count)
Equivalent to `valueOf(char[])`.

boolean endsWith(String suffix)
Tests if this string ends with the specified suffix.

boolean equals(Object anObject)
Compares this string to the specified object.

boolean equalsIgnoreCase(String anotherString)
Compares this `String` to another `String`, ignoring case considerations.

static String format(Locale l, String format, Object... args)
Returns a formatted string using the specified locale, format string, and arguments.

static String format(String format, Object... args)
Returns a formatted string using the specified format string and arguments.

byte[] getBytes()
Encodes this `String` into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

byte[] getBytes(Charset charset)
Encodes this `String` into a sequence of bytes using the given `charset`, storing the result into a new byte array.

void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)
Deprecated.
This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the `getBytes()` method, which uses the platform's default charset.

byte[] getBytes(String charsetName)
Encodes this `String` into a sequence of bytes using the named charset, storing the result into a new byte array.

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

int hashCode()
Returns a hash code for this string.

int indexOf(int ch)
Returns the index within this string of the first occurrence of the specified character.

int indexOf(int ch, int fromIndex)
Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

int indexOf(String str)
Returns the index within this string of the first occurrence of the specified substring.

int indexOf(String str, int fromIndex)
Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

String intern()
Returns a canonical representation for the string object.

boolean isEmpty()
Returns true if, and only if, `length()` is 0.

static String join(CharSequence delimiter, CharSequence... elements)
Returns a new `String` composed of copies of the `CharSequence` elements joined together with a copy of the specified `delimiter`.

static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)
Returns a new `String` composed of copies of the `CharSequence` elements joined together with a copy of the specified `delimiter`.

int lastIndexOf(int ch)
Returns the index within this string of the last occurrence of the specified character.

int lastIndexOf(int ch, int fromIndex)
Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int

`lastIndexOf(String str)`

Returns the index within this string of the last occurrence of the specified substring.

int

`lastIndexOf(String str, int fromIndex)`

Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

int

`length()`

Returns the length of this string.

boolean

`matches(String regex)`

Tells whether or not this string matches the given regular expression.

int

`offsetByCodePoints(int index, int codePointOffset)`

Returns the index within this String that is offset from the given index by codePointOffset code points.

boolean

`regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)`

Tests if two string regions are equal.

boolean

`regionMatches(int toffset, String other, int ooffset, int len)`

Tests if two string regions are equal.

String

`replace(char oldChar, char newChar)`

Returns a string resulting from replacing all occurrences of oldChar in this string with newChar.

String

`replace(CharSequence target, CharSequence replacement)`

Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.

String

`replaceAll(String regex, String replacement)`

Replaces each substring of this string that matches the given regular expression with the given replacement.

String

`replaceFirst(String regex, String replacement)`

Replaces the first substring of this string that matches the given regular expression with the given replacement.

String

`split(String regex)`

Splits this string around matches of the given regular expression.

String[]

`split(String regex, int limit)`

Splits this string around matches of the given regular expression.

boolean

`startsWith(String prefix)`

Tests if this string starts with the specified prefix.

boolean

`startsWith(String prefix, int toffset)`

Tests if the substring of this string beginning at the specified index starts with the specified prefix.

CharSequence

`subSequence(int beginIndex, int endIndex)`

Returns a character sequence that is a subsequence of this sequence.

String

`substring(int beginIndex)`

Returns a string that is a substring of this string.

String

`substring(int beginIndex, int endIndex)`

Returns a string that is a substring of this string.

char[]

`toCharArray()`

Converts this string to a new character array.

String

`toLowerCase()`

Converts all of the characters in this String to lower case using the rules of the default locale.

String

`toLowerCase(Locale locale)`

Converts all of the characters in this String to lower case using the rules of the given Locale.

String

`toString()`

This object (which is already a string!) is itself returned.

String

`toUpperCase()`

Converts all of the characters in this String to upper case using the rules of the default locale.

String

`toUpperCase(Locale locale)`

Converts all of the characters in this String to upper case using the rules of the given Locale.
String

trim()
Returns a string whose value is this string, with any leading and trailing whitespace removed.

static String
valueOf(boolean b)
Returns the string representation of the boolean argument.

static String
valueOf(char c)
Returns the string representation of the char argument.

static String
valueOf(char[] data)
Returns the string representation of the char array argument.

static String
valueOf(char[], int offset, int count)
Returns the string representation of a specific subarray of the char array argument.

static String
valueOf(double d)
Returns the string representation of the double argument.

static String
valueOf(float f)
Returns the string representation of the float argument.

static String
valueOf(int i)
Returns the string representation of the int argument.

static String
valueOf(long l)
Returns the string representation of the long argument.

static String
valueOf(Object obj)
Returns the string representation of the Object argument.

- Methods inherited from class java.lang.Object
  clone, finalize, getClass, notify, notifyAll, wait, wait, wait

- Methods inherited from interface java.lang.CharSequence
  char, codePoints

lastIndexOf

public int lastIndexOf(int ch)
Returns the index within this string of the last occurrence of the specified character. For values of ch in the range from 0 to 0xFFFF (inclusive), the index (in Unicode code units) returned is the largest value k such that:
this.charAt(k) == ch
is true. For other values of ch, it is the largest value k such that:
this.codePointAt(k) == ch
is true. In either case, if no such character occurs in this string, then -1 is returned. The String is searched backwards starting at the last character.

Parameters:
  ch - a character (Unicode code point).

Returns:
  the index of the last occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur.