1 Lexical Structure

1.1 Unicode

Before the Java source program is tokenized, each character is converted from the native character set to the corresponding Unicode character and character sequences of the form `\udddd`, where `d` is a hexadecimal digit, are converted to the corresponding Unicode character `U+dddd`. Some example Unicode characters:

<table>
<thead>
<tr>
<th>escape</th>
<th>Unicode character name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\u007B</td>
<td>{</td>
</tr>
<tr>
<td>\u00D7</td>
<td>×</td>
</tr>
<tr>
<td>\u015C</td>
<td>S</td>
</tr>
<tr>
<td>\u0635</td>
<td>ء</td>
</tr>
<tr>
<td>\u03A9</td>
<td>Ω</td>
</tr>
<tr>
<td>\u201C</td>
<td>“</td>
</tr>
<tr>
<td>\u5C71</td>
<td>山</td>
</tr>
</tbody>
</table>

1.2 Comments

```java
// α) A comment to the end of the line.

/** β) A traditional comment as in C, C++, and #. */ /* These comments don’t nest! */
/** γ) “Documentation” comments may contain HTML
   and ’javadoc’ tags: @author @see @return @throws */
```

1.3 Keywords

```java
abstract assert boolean break byte case catch char class const continue default do double else enum extends finally for goto if implements import instanceof int interface long native new package private protected public return short static strictfp super switch synchronized this throw throws transient try void volatile while
```

The keywords `const` and `goto` not used in the Java grammar. The literals `true`, `false`, and `null` are reserved words in the Java syntax.

1.4 Literals

Besides the literals `true`, `false` for boolean values, there are several other kinds of literals in Java. The literal `null` is a distinguished value that belongs to all object types.

Character literals are delimited by the single quote and string literals are delimited by the double quote character. Character escapes are present only in character and string literals. Unicode escapes are not the same as character escapes. The complete list of character escapes follows:

<table>
<thead>
<tr>
<th>escape</th>
<th>Unicode character name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>U+0008 backspace</td>
</tr>
<tr>
<td>\t</td>
<td>U+0009 horizontal tabulation (HT), tab</td>
</tr>
<tr>
<td>\n</td>
<td>U+000A line feed (LF) aka newline</td>
</tr>
<tr>
<td>\f</td>
<td>U+000C form feed (FF)</td>
</tr>
<tr>
<td>\r</td>
<td>U+000D carriage return (CR)</td>
</tr>
<tr>
<td>&quot;</td>
<td>U+0022 QUOTATION MARK (double quote)</td>
</tr>
<tr>
<td>'</td>
<td>U+0027 APOSTROPHE (single quote)</td>
</tr>
<tr>
<td>\</td>
<td>U+005C REVERSE SOLIDUS (backslash)</td>
</tr>
</tbody>
</table>

2 Data

2.1 Primitive Data Types

Java has eight fundamental units of data:

- Boolean. `boolean`: two-valued type with elements `true` and `false`
- Character. `char`: values from Unicode (UTF-16)
- Integral (twos complement). `byte`, `short`, `int`, `long`
- Floating point (IEEE754). `float`, `double`

All other data types (strings, arrays, user-defined classes, classes in the Java API, etc.) are `objects` and allocated on the heap and reclaimed implicitly by the garbage collector. Java has no unsigned byte or word type.

2.2 Wrapper Classes

Corresponding to each primitive data type is a class in the package `java.lang: Boolean, Character, Byte, Short, Integer, Long, Float, and Double. These classes contain, box, or wrap a value of the corresponding type. Java automatically boxes and unboxes all the primitive data types blurring the distinction between primitive and non-primitive data. This is especially valuable in the context of generics in which all data must be treated as an object.

The wrapper classes have static methods (e.g., `parseInt()`), and fields (e.g., `MAX_VALUE`), specific to the individual primitive types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>Integer.MAX_VALUE</code></td>
</tr>
<tr>
<td></td>
<td><code>0x7fffffff</code></td>
</tr>
<tr>
<td><code>long</code></td>
<td><code>Long.MAX_VALUE</code></td>
</tr>
<tr>
<td></td>
<td><code>0x1fffffffffffffffL</code></td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>Float.MAX_VALUE</code></td>
</tr>
<tr>
<td></td>
<td><code>0x1.fffffP+127f</code></td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>Double.MAX_VALUE</code></td>
</tr>
<tr>
<td></td>
<td><code>0x1.fffffffffffffffP+1023</code></td>
</tr>
<tr>
<td></td>
<td><code>2^31 − 1 ≈ 2.1 × 10^9</code></td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>Integer.MIN_VALUE</code></td>
</tr>
<tr>
<td></td>
<td><code>0x80000000</code></td>
</tr>
<tr>
<td><code>long</code></td>
<td><code>Long.MIN_VALUE</code></td>
</tr>
<tr>
<td></td>
<td><code>0x8000000000000000L</code></td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>Float.MIN_VALUE</code></td>
</tr>
<tr>
<td></td>
<td><code>0.000002p-126f</code></td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>Double.MIN_VALUE</code></td>
</tr>
<tr>
<td></td>
<td><code>0.00000000000001P-1022</code></td>
</tr>
<tr>
<td></td>
<td><code>1.4 × 10^-45</code></td>
</tr>
<tr>
<td></td>
<td><code>3.4 × 10^38</code></td>
</tr>
<tr>
<td></td>
<td><code>1.8 × 10^308</code></td>
</tr>
<tr>
<td></td>
<td><code>4.9 × 10^-324</code></td>
</tr>
</tbody>
</table>
2.3 java.lang.String

Though not a primitive data type, java.lang.String is an important data type with special syntax (the double quotes) and a predefined infix operation (concatenation):

"string literal" + " more"

A string is an immutable sequence of characters with the following as some of the more common methods.

```java
int length()                        // string length
char charAt(int i)                  // i-th character
String substring(int i, int j)     // i-th to j-th characters
int indexof(String s)              // s is a substring
boolean equals(Object o)           // lexicographic ordering
int compareTo(String s)
```

The class java.lang.StringBuilder is a mutable sequence of characters with many of the same methods as java.lang.String. It implements the interface java.lang.CharSequence, but not java.lang.Comparable<StringBuilder>.

2.4 Arrays

Fixed-length arrays have special syntax in Java. For variable-length (dynamic) structures see the generic class java.util.ArrayList. The type of an array is written like this:

```java
int[]          
String[]       
```

Like all objects an array can be created using the keyword `new` like this:

```java
new int[4]     
new String[7]  
```

Elements of an array can be accessed only with integer indices starting with zero: `a[0]` or `b[3]`. The length of an array is obtained like this: `a.length` or `b.length` (a common mistake is to add () like for String). Strictly speaking there are no multi-dimensional arrays (like in Fortran) — only arrays of arrays as in C, and C++.

Important static methods from java.util.Arrays include:

```java
String toString (Type[] a)
int hashCode (Type[] a)
boolean equals (Type[] a, Type[] other)
int binarySearch (Type[] a, Type key)
int binarySearch (Type[] a, int from, int to, Type key)
void fill (Type[] a, Type val)
void sort (Type[] a)
void sort (Type[] a, int from, int to)
Type[] copyOf (Type[] a, int new_length)
Type[] copyOfRange (Type[] a, int from, int to)
```

where `Type` is a primitive type. Versions of these function for arrays of objects are also in java.util.Arrays.

2.5 Objects

The keyword `new` creates a new object, a new instance of the class in the heap. Some examples follow:

```java
Date now = new Date ();
Random rng = new Random ();
Color c = new Color (0, 127, 255);
StringBuilder s = new StringBuilder ();
Number n = new BigDecimal (3);
Scanner stdin = new Scanner (System.out). useDelimiter ();
```

Strings and arrays are constructed implicitly by literals with special syntax; `new String("abc")` is legal, but redundant.

```java
String s = "abc";
int[] a1 = {1,2}; // 'new' optional in array decl
int[][] a2 = {{0},{0,1},{0,1,2}};
```

However, the keyword `new` is required in array literals not in a declaration:

```java
int x = binarySearch (new int[] {1,4,6,7,9}, 5);
```

Each new object is a different entity in the heap. So

```java
new BigDecimal (1) == new BigDecimal (1)
```

yields false, but `equals()` inherited from Object

```java
new BigDecimal (1) . equals (new BigDecimal (1))
```

yields true.

2.6 Enumerated Types

Simple enumerated types provide a convenient way to represent data which has a small set of fixed, discrete values.

```java
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT }
```

Enumerated types are a particular kind of class with two static, generated methods, with the following types illustrated with the class `Suit`:

```java
Suit[] values();
Suit valueOf (String name);
```

and a common superclass java.lang.Enum<E extends Enum<E>> with methods

```java
int compareTo (E o)
int ordinal ()
```
enum Direction { NORTH (0,+1), EAST (+1,0), SOUTH (0,-1), WEST (-1,0); 
private final int x, y; 
private Direction (int x, int y) { this.x=x; this.y=y; } 
public Direction rotate (int i) { 
assert i>=0; // Caution: % and negative numbers 
return Direction.values()[((this.ordinal()+i)%4)]; 
}
}

Java provides the support classes EnumSet and EnumHashMap which are used as in the following example:

for (Day d: EnumSet.range (Day.MON, Day.FRI) {} 
EnumSet<Style> set = 
EnumSet.of (Style.BOLD, Style.ITALIC);

3 Expressions

3.1 Test for equality

Java has the binary, infix operator == for testing the equality of values of primitive data types. For objects, the method equals() is usually what is needed. “Not equals” is !=.

String word = n!=1 ? "cats" : "cat";

3.2 Binary, infix operators

<table>
<thead>
<tr>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>* / %</td>
<td>num. mult., div., remainder</td>
</tr>
<tr>
<td>+ -</td>
<td>num. addition subtraction</td>
</tr>
<tr>
<td>+</td>
<td>string concatenation</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>integral left shift</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>integral r. shift with sign extension</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;</td>
<td>integral r. shift with zero extension</td>
</tr>
<tr>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>num. less/greater than (or equal)</td>
</tr>
</tbody>
</table>

3.3 Boolean Operations

The boolean data type has three operators: !, &&, and ||, for the three familiar operations: negation, conjunction, and disjunction.

3.4 java.lang.Math

The java.lang.Math class has many static members. These members can be called using their simple (i.e., unqualified) name by using the following import declaration:

import static java.lang.Math.*;

double abs(double x) absolute value of x
double max(double x, double y) maximum of x and y
double min(double x, double y) minimum of x and y
double sqrt(double x) square root √x
double cbrt(double x) cube root 3√x
double pow(double x, double y) x^y
double atan2(double x, double y) angle of x and y
double hypot(double x, double y) √x^2+y^2
double ceil(double x) closest long

double sin(double theta) sine function
double cos(double theta) cosine function
double tan(double theta) tangent function
double asin(double theta) inverse sine function
double acos(double theta) inverse cosine function
double atan(double theta) inverse tangent function

double toRadians(double deg) and double toDegrees(double rad) to convert back and forth.

3.5 java.math.BigInteger and Bitwise Operations

The immutable class BigInteger supports arbitrary-precision integers.

Arithmetic methods:

BigInteger abs() 
BigInteger max(BigInteger b) 
BigInteger min(BigInteger b) 
BigInteger negate() - this 
BigInteger add(BigInteger b) this + b 
BigInteger subtract(BigInteger b) this - b 
BigInteger multiply(BigInteger b) this * b 
BigInteger pow(BigInteger b) 
BigInteger mod(BigInteger b) (b + this%b)%b 
BigInteger remainder(BigInteger b) this % b 
BigInteger divide(BigInteger b) this / b 
BigInteger[] divideAndRemainder(BigInteger b)

BigInteger gcd(BigInteger b)
Set or logic methods:

- BigInteger and(BigInteger b) \( \text{this} \land b \) intersection/conjunction
- BigInteger or(BigInteger b) \( \text{this} \lor b \) union/disjunction
- BigInteger xor(BigInteger b) \( \text{this} \oplus b \) symmetric difference
- BigInteger andNot(BigInteger b) \( \text{this} \land \neg b \) set minus

Bitwise methods:

- BigInteger not() \( \neg \text{this} \)
- BigInteger shiftLeft(int n) \( \text{this} << n \)
- BigInteger shiftRight(int n) \( \text{this} >> n \)
- BigInteger clearBit(int n) \( \text{this} \land \neg (1 << n) \)
- BigInteger flipBit(int n) \( \text{this} \oplus (1 << n) \)
- BigInteger setBit(int n) \( \text{this} \lor (1 << n) \)
- boolean testBit(int n) \( \text{this} \land (1 << n) \neq 0 \)

4 Statements

Templates for some compound statements.

```java
if ( /* condition */ ) {
    /* statements executed if true */
}
```

```java
if ( /* condition */ ) {
    /* statements executed if true */
} else {
    /* statements executed if false */
}
```

```java
if ( /* condition1 */ ) {
    /* statements executed if true*/
} else if ( /* condition2 */ ) {
    /* statements */
} else if ( /* condition3 */ ) {
    /* statements */
} else {
    /* statements executed if none of the above*/
}
```

```java
for (int index=0; index<a.length; index++) {
    /* statements */
}
```

```java
for (int v: a) {
    /* statements */
}
```

```java
for (;;) {
    /* statements */
    if (/* condition */ ) break;
    /* statements */
}
```

while ( /* condition */ ) {
    /* statements */
}

5 Functions and Procedures

5.1 Method Syntax

```java
<method declaration> ::= 
    [<type parameter list>] <type> <identifier> 
    "(" [ <formal parameters list> ] 
    ")" 
    ["throws" <qualified identifier list>] 
    "{" <method body> ""}
```

5.2 Varargs

Methods can have a variable number of arguments (varargs) in Java. The three periods after the parameter’s type indicate that the parameter may be passed as an array or as a sequence of zero or more actual arguments of that type. Varargs can be used only in the final argument position.

Here are some method declarations in the Java API’s using varargs:

```java
PrintWriter printf (String format, Object... args)
Method getMethod (String name, Class<?>... parameterTypes)
Object invoke (Object obj, Object... args)
```

A varargs, formal parameter is accessed exactly as if it were an array.

```java
public class Test {
    public static void main (String... args) {
        int pass=0, fail=0;
        for (String className: args) {
            try {
                final Class c = Class.forName (className);
                c.getMethod("test").invoke(c.newInstance());
                pass++;
            } catch (Exception ex) {
                System.out.printf ("%s failed.%n", className);
                fail++;
            }
        }
        System.out.printf ("pass=%d; fail=%d%n", pass, fail);
    }
}
```
6 Input/Output

6.1 Standard IO

Java.lang.System has static fields for distinguished input and output streams:

<table>
<thead>
<tr>
<th>InputStream in</th>
<th>standard input stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrintStream out</td>
<td>standard output stream</td>
</tr>
<tr>
<td>PrintStream err</td>
<td>standard error output stream</td>
</tr>
</tbody>
</table>

Normally the input stream is associated with the keyboard and the output stream is associated with the display device.

The class InputStream has a method int read() to read the next byte of data. Often it is convenient to use an InputStream with the class java.util.Scanner which has high-level methods for reading text (see the next subsection).

The class PrintStream has high-level methods for printing Java data types as text and two synonymous methods for C-style formatting of data: printf and format.

6.2 java.util.Scanner

The java.util.Scanner is useful in reading and converting input text into Java data types. By default, the input text is delimited by \"\p{javaWhiteSpace}+\".

boolean hasNext() String next()
boolean hasNext(Type x) Type nextType()

The boolean function hasNext(Type x) determines if next token in the stream is of that type, and nextType(x) returns the next token expecting it to be of that type. Type is one of Byte, Short, Int, Long, Float, Double, Boolean, BigInteger, BigDecimal, but not Char. The function returns a value of the appropriate Java data type: byte, short, int, long, float, double, boolean, BigInteger, BigDecimal.

hasNext(java.util.regex.Pattern x) determines if the next delimited token matches the regular expression x. And next(java.util.regex.Pattern x) returns the string that matches.

The methods skip(Pattern x) and findInHorizon(Pattern x, int h) of the java.util.Scanner class ignore delimiters. To get the next character:

char nextChar = scan.findInHorizon(".(?s)" ,1).charAt(0)

Using the empty string as a delimiter to the scanner will result in each token (string) being a single character long. Using the Java string "\\n" as a delimiter will result in one token (string) consisting of the entire input.

6.3 java.util.Formatter

Format specifiers have the following syntax:

% [index \$ ] [options] [width] [ . precision ] format

where index is a decimal integer indicating which argument 1$, 2$, etc. and format is one of the following characters:

b B h H s S c C d o x X f G g A t T % n

<table>
<thead>
<tr>
<th>boolean</th>
<th>byte, Byte, short, Short, int, Integer, long, Long, BigInteger</th>
</tr>
</thead>
<tbody>
<tr>
<td>integral</td>
<td>byte, Byte, short, Short, int, Integer</td>
</tr>
<tr>
<td>floating point</td>
<td>float, Float, double, Double, BigDecimal</td>
</tr>
<tr>
<td>character</td>
<td>char, Character, byte, Byte, short, Short, int, Integer</td>
</tr>
<tr>
<td>date/time</td>
<td>long, Long, Calendar, Date</td>
</tr>
<tr>
<td>percent</td>
<td>% needs no argument</td>
</tr>
<tr>
<td>newline</td>
<td>\n needs no argument</td>
</tr>
</tbody>
</table>

d 1512 "%10d" 1512

d 1512 "%1,10d" 1512

f 159.16810 "%10.2f" 159.17

e 159.16810 "%10.1e" +1.5e02

e 159.16810 "%10.5g" 159.17

s "hello" "%10s" hello

s "hello" "%-10s" "hello"

s "hello" "%-10.4s" "hello"

`C' "%c" "A"

7 Simple Program

```
import java.util.Scanner;

public final class Sum {

    public static void main(String[] args) {
        final Scanner stdin = new Scanner(System.in);

        int sum = 0;

        while (stdin.hasNextInt()) {
            sum += stdin.nextInt();
        }

        System.out.format("%sum=%d, \d%n", sum);
    }
}
```

Compiling and running with the Sun JDK:
There are many IDEs: Emacs, BlueJ, Eclipse, Netbeans

8 Object-Oriented Programming

8.1 Class

A class declaration begins with the word class:

```plaintext
<class declaration> ::= 
"class" <identifier> 
{" <class body> "}

<class body> ::= { <class body declaration> }
<class body declaration> ::= ";"
/ "static" <statement block>
/ { <modifier> } <member declaration>
<member declaration> ::= <field declaration>
/ <method declaration>
/ <constructor declaration>
/ <interface declaration>
/ <class declaration>
```

Note that classes can be nested and that they may contain static initialization blocks.

8.2 Members

Static members of classes (fields and methods) should be accessed using the class name. (The class name may be omitted if access is from within the same class.) For example,

```plaintext
System.out // access static field 'out' in class System
Math.abs (3.4) // invoke static method 'abs' in class Math
String.format () // invoke static method
Arrays.toString(new int[]{1,2,3})
```

Non-static members of classes (fields and methods) may be accessed only via an instance of the class.

```plaintext
stdin.nextInt() // where stdin is an instance of Scanner
out.printf () // where out is an instance of PrintWriter
```

(If the instance is omitted, this is assumed.) By using the style convention that class names are capitalized, static member access is easily distinguished from non-static access.

- `private`—members declared `private` are accessible within the class itself.
- "package"—members declared with no access modifier are accessible in classes in the same package.
- `protected`—members declared `protected` are accessible in subclasses (in the same package or not) and in the class itself.
- `public`—members declared `public` are accessible anywhere the class is accessible.

<table>
<thead>
<tr>
<th>access from</th>
<th>private</th>
<th>“package”</th>
<th>protected</th>
<th>public</th>
</tr>
</thead>
<tbody>
<tr>
<td>same class</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>same package</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>in subclass, out of package</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>non-subclass, out of package</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

8.3 Inheritance

A subclass is created by naming the unique superclass. A subclass has only one superclass. If a superclass is not explicitly named, the class Object is the superclass. So, the class Object is the unique root of the Java subclass hierarchy. This has the importance consequence that all objects inherit from Object.

```plaintext
<class declaration> ::= 
"class" <identifier> 
["extends" <type>]
{" <class body> "}
```

Some of the instantiable subclasses of Object are:

- `java.io.File`
- `java.util.Date`
- `java.util.Scanner`
- `java.lang.String`
- `java.lang.StringBuilder`
- `java.lang.Exception`
- `java.lang.Thread`
8.4 Overriding

One predefined annotation is useful for marking overridden methods:

```java
@Override
public String toString() {
    return String.format("representation");
}
```

8.5 Abstract Classes

A class declaration may be modified with the keyword `abstract` marking it as an uninstantiable, abstract class. These classes can have abstract methods:
methods declared but with no body of code. Classes inheriting from an abstract class have the responsibility to implement the abstract methods.

8.6 Interfaces

Interfaces are all abstract classes. Since no code can be inherited from an interface, a class can implement as many interfaces as desired.

```java
<class declaration> ::= 
    "class" <identifier> 
    ["extends" <type>]  
    ["implements" <type list>] 
    "{" <class body> "}"
```

Two important interfaces are `Comparable<T>` and `Comparator<T>`.

8.7 Dynamic Dispatch

The following program prints “X” because the method invoked by the call `o.toString()` is dynamically determined by the class it refers to (class `X`) and not statically by the type of the variable `o` (class `Object`).

```java
class X extends Object {
    @Override
    public String toString() { return "X";
    public static void main (String[] args) {
        Object o = new X();
        System.out.println (o.toString());
    }
}
```

9 Exceptions

A checked exception is one that extends the class `java.lang.Exception`, but does not extend the class `java.lang.RuntimeException`. Typical (unchecked) runtime exceptions are `NullPointerException` and `ArrayIndexOutOfBoundsException`. These exceptions generally indicate logical errors in the program.

```java
try {
    // block monitored for exceptions
} catch (IOException ex) {
    ex.printStackTrace();
} finally {
    // executed no matter what
}
```

9.1 Assertions

The `assert` statement is ignored unless assertion checking is enabled with java command line options:

```java
java -ea -da:... -da:pack.subpack... pack.Main
```

When enabled, the statement `assert expr: str` acts like:

```java
if(!expr) throw new AssertionError(str);
```

The following lines make use of assertions to clarify the computation of the integer square root of `n`:

```java
assert 0<n : "precond";
int r = 1;
while (n>=(r+1)*(r+1)) {
    assert r*r<=n : "loop inv";
    r++;
}
assert r*r<=n && n<(r+1)*(r+1);
```

10 Generics

Java has generic classes and methods. The following is the syntax of a generic class.

```java
<class declaration> ::= 
    "class" <identifier> [<type parameter list>] 
    ["extends" <type>]  
    ["implements" <type list>] 
    "{" <class body> "}"
```
11 Collection Classes

11.1 Sequences
ArrayList<
>, LinkedList<
>: List<
> get(i), set(i,e), add(i,e), remove(i)

11.2 Stacks, Queues, Dequeues
ArrayDeque<
>, LinkedList<
>: Deque<
> addFirst(e), addLast(e), removeFirst(e), removeLast(e)

11.3 Sets
HashSet<
>: Set<
> add(e), contains(e)

Set <String > set =
new HashSet <String >( Arrays . asList ("a", "an", "the"));

11.4 Dictionaries or Maps
HashMap<K,V>: Map<K,V> put(k,v), get(k), containsKey(k),

TreeMap<K,V>: NavigableMap<K,V> : Map<K,V> put(k,v), get(k), containsKey(k),

12 Regular Expressions java.util.regex

Regular expressions denote patterns of strings and can be represented in Java by special string literals. Regular expressions use \ to escape meta-characters. Character escapes also start with \\ . One must escape the escape character to write regular expressions as Java strings—the backslash appears twice.

"\\Qa"b\E[\p{Punct}&&[^-]]"

Predefined character classes.
\d [0-9] a digit
\s [ \t\n\x0B\f\r] whitespace character
\w [a-zA-Z0-9_] character of “word”

Character classes from POSIX, java.lang.Character, Unicode blocks and Unicode categories.
\p{Lower} [a-z] lower-case letter
\p{Upper} [A-Z] upper-case letter
\p{ASCII} [\x00-\x7F] all ASCII chars
\p{Digit} [0-9] same as \d
\p{Alpha} [\p{Lower}\p{Upper}] upper and lower case
\p{Alnum} [\p{Alpha}\p{Digit}] letters and digits
\p{Space} [ \t\n\x0B\f\r] same as \s
\p{XDigit} [0-9a-fA-F] hexadecimal digit
\p{Cntrl} [\X00-\X1F\x7F] control character

r? zero or one
r* zero or more
r+ one or more
r{n} exactly n times
r{n,} n times or more
r{n,m} at least n and no more than m

All unmodified quantifiers are greedy, meaning they match as many elements of the string as possible without causing the overall match to fail. In contrast, a reluctant quantifier will match as few elements of the strings as possible. You can make a quantifier reluctant by adding the ‘?’ character.

A possessive quantifier will match as much of the target as possible (like greedy) even if it means that the remainder of the regex will fail. Backtracking is prevented (improving efficiency), just like independent (aka atomic) grouping. You can make a quantifier possessive by adding the ‘+’ character.

(r) capturing group by position; refered to later using \d
(?<name>r) capturing group by name; refered to later using \k<name>
(?!r) simple group (no capturing, no back reference to it)
(?<=r) an independent (no backtracking), non-capturing group

Zero-width (non-capturing) assertions.

<table>
<thead>
<tr>
<th>positive</th>
<th>negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>ahead</td>
<td>(?=r)</td>
</tr>
<tr>
<td>behind</td>
<td>(?&lt;=r)</td>
</tr>
</tbody>
</table>