Some New Features in Java 5.0

Light Version

Advanced Topics in Java

Khalid Azim Mughal
khalid@ii.uib.no
http://www.ii.uib.no/~khalid

Version date: 2006-01-16

Overview

- Enumerated types
- Automatic Boxing and Unboxing of Primitive Values
- Enhanced for loop
- Static Import
- Formatted Output
- Formatted Input

ENUMS

Enumerated Types

- An enumerated type defines a *finite set of symbolic names and their values*.
- Standard approach is the `int` *enum pattern* (or the analogous `String` *enum pattern*):

```
public class MachineState {
  public static final int BUSY = 1;
  public static final int IDLE = 0;
  public static final int BLOCKED = -1;
  //...
}
```

```
public class Machine {
  int state;
  public void setState(int state) {
    this.state = state;
  }
  //...
}
```

```
public class IntEnumPatternClient {
  public static void main(String[] args) {
    Machine machine = new Machine();
    machine.setState(MachineState.BUSY);   // (1) Constant qualified by class name
    machine.setState(1);                    // Same as (1)
    machine.setState(5);                    // Any int will do.
    System.out.println(MachineState.BUSY); // Prints "1", not "BUSY".
  }
}
```

Some Disadvantages of the int Enum Pattern

- Not typesafe.
  - Any `int` value can be passed to the `setState()` method.
- No namespace.
  - A constant must be qualified by the class (or interface) name, unless the class is extended (or the interface is implemented).
- Uninformative textual representation.
  - Only the value can be printed, not the name.
- Constants compiled into clients.
  - Clients need recompiling if the constant values change.

Typesafe Enum Construct

- The enum construct provides support for enum types:

```
enum MachineState { BUSY, IDLE, BLOCKED } // Canonical form
enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

- Keyword `enum` is used to declare an *enum type*.
- Overcomes all the disadvantages of the `int` enum pattern, and is more powerful!

Properties of the Enum Type

- An *enum declaration* is a special kind of class declaration:
  - It can be declared at the top-level and as static enum declaration.

```
// (1) Top level enum declaration
public enum SimpleMeal
    { BREAKFAST, LUNCH, DINNER }
```

```
public class EnumTypeDeclarations {
  // (2) Static enum declaration is OK.
  public enum SimpleMeal
      { BREAKFAST, LUNCH, DINNER };
  public void foo() {
    // (3) Local (inner) enum declaration is NOT OK!
    enum SimpleMeal
      { BREAKFAST, LUNCH, DINNER }
  }
}
```


Enum Constructors

- Each constant declaration can be followed by an argument list that is passed to the constructor of the enum type having the matching parameter signature.
  - An implicit standard constructor is created if no constructors are provided for the enum type.
  - As an enum cannot be instantiated using the new operator, the constructors cannot be called explicitly.

```
public enum Meal {
  BREAKFAST(7,30), LUNCH(12,15), DINNER(19,45);
  Meal(int hh, int mm) {
    assert (hh >= 0 && hh <= 23): "Illegal hour.";
    assert (mm >= 0 && hh <= 59): "Illegal mins.";
    this.hh = hh;
    this.mm = mm;
  }
  // Time for the meal.
  private int hh;
  private int mm;
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }
}
```

Methods Provided for the Enum Types

- Names of members declared in an enum type cannot conflict with automatically generated member names:
  - The enum constant names cannot be redeclared.
  - The following methods cannot be redeclared:

`static <this enum class>[] values()`	Returns an array containing the constants of this enum class, in the order they are declared.
`static <this enum class> valueOf(String name)`	Return the enum constant with the specified name

- Enum types are based on the `java.lang.Enum` class which provides the default behavior.
- Enums cannot declare methods which override the final methods of the `java.lang.Enum` class:
  - `clone()`, `compareTo(Object)`, `equals(Object)`, `getDeclaringClass()`, `hashCode()`, `name()`, `ordinal()`.
  - The final methods do what their names imply, but the `clone()` method throws an `CloneNotSupportedException`, as an enum constant cannot be cloned.

- *Note that the enum constants must be declared before any other declarations in an enum type.*

```
public class MealClient {
  public static void main(String[] args) {

    for (Meal meal : Meal.values())
      System.out.println(meal + " served at " +
                         meal.getHour() + ":" + meal.getMins() +
                         ", has the ordinal value " +
                         meal.ordinal());
  }
}
```

Output from the program:

```
BREAKFAST served at 7:30, has the ordinal value 0
LUNCH served at 12:15, has the ordinal value 1
DINNER served at 19:45, has the ordinal value 2
```

Enums in a `switch` statement

- The `switch` expression can be of an enum type, and the `case` labels can be enum constants of this enum type.

```
public class EnumClient {
  public static void main(String[] args) {

    Machine machine = new Machine();
    machine.setState(MachineState.IDLE);
    // ...
    MachineState state = machine.getState();
    switch(state) {
    //case MachineState.BUSY:// Compile error: Must be unqualified.
      case BUSY:    System.out.println(state + ": Try later.");        break;
      case IDLE:    System.out.println(state + ": At your service."); break;
      case BLOCKED: System.out.println(state + ": Waiting on input."); break;
    //case 2:               // Compile error: Not unqualified enum constant.
      default: assert false: "Unknown machine state: " + state;
    }
  }
}
```


Declaring New Members in Enum Types

- Any *class body declarations* in an enum declaration apply to the enum type exactly as if they had been present in the class body of an ordinary class declaration.

AUTOMATIC BOXING/UNBOXING

Automatic Boxing and Unboxing of Primitive Values

```
int     i    = 10;
Integer iRef = new Integer(i);     // Explicit Boxing
int     j    = iRef.intValue();    // Explicit Unboxing

        iRef = i;                  // Automatic Boxing
        j    = iRef;               // Automatic Unboxing
```

- Automatic boxing and unboxing conversions alleviate the drudgery in converting values of primitive types to objects of the corresponding wrapper classes and vice versa.
- *Boxing conversion* converts primitive values to objects of corresponding wrapper types.
- *Unboxing conversion* converts objects of wrapper types to values of corresponding primitive types.

Automatic Boxing and Unboxing Contexts

- Assignment Conversions on boolean and numeric types.

```
boolean boolVal = true;
byte  b = 2;
short s = 2;
char  c ='2';
int   i = 2;

// Boxing
Boolean   boolRef = boolVal;
Byte      bRef = (byte) 2;  // cast required as int not assignable to Byte
Short     sRef = (short) 2; // cast required as int not assignable to Short
Character cRef = '2';
Integer   iRef = 2;
// Integer   iRef1 = s;   // short not assignable to Integer

// Unboxing
boolean boolVal1 = boolRef;
byte  b1 = bRef;
short s1 = sRef;
char  c1 = cRef;
int   i1 = iRef;
```


- Method Invocation Conversions on actual parameters.

```
...
flipFlop("(String, Integer, int)", new Integer(4), 2004);
...

private static void flipFlop(String str, int i, Integer iRef) {
    out.println(str + " ==> (String, int, Integer)");
}
```

Output:

```
(String, Integer, int) ==> (String, int, Integer)
```

- Casting Conversions:

```
Integer iRef = (Integer) 2;  // Boxing followed by identity cast
int     i    = (int) iRef;   // Unboxing followed by identity cast
// Long    lRef = (Long) 2;     // int not convertible to Long
```

- Numeric Promotion: Unary and Binary

```
Integer iRef = 2;
long    l1   = 2000L + iRef; // Binary Numeric Promotion
int     i    = -iRef;        // Unary Numeric Promotion
```

- In the `if` statement, condition can be `Boolean`:

```
Boolean expr = true;
if (expr)
  out.println(expr);
else
  out.println(!expr);  // Logical complement operator
```

- In the `switch` statement, the `switch` expression can be `Character`, `Byte`, `Short` or `Integer`.

```
// Constants
final short ONE     = 1;
final short ZERO    = 0;
final short NEG_ONE = -1;

// int expr = 1;       // (1) short is assignable to int. switch works.
// Integer expr = 1;   // (2) short is not assignable to Integer. switch compile error.
Short expr = (short)1; // (3) Cast required even though value is in range.
switch (expr) {        // (4) expr unboxed before case comparison.
  case ONE:     out.println(">="); break;
  case ZERO:    out.println("=="); break;
  case NEG_ONE: out.println("<="); break;
  default: assert false;
}
```


- In the `while`, `do-while` and `for` statements, the condition can be `Boolean`.

```
Boolean expr = true;
while (expr)
  expr = !expr;

Character[] version = {'1', '.', '5'};        // Assignment: boxing
for (Integer iRef = 0;                        // Assignment: boxing
     iRef < version.length;                   // Comparison: unboxing
     ++iRef)                                  // ++: unboxing and boxing
  out.println(iRef + ": " + version[iRef]);   // Array index: unboxing
```

ENHANCED FOR LOOP

Enhanced `for` Loop: `for(:)`

- The `for(:)` loop is designed for iteration over *array and collections*.

Iterating over Arrays:

`for(;;)` Loop	`for(:)` Loop

`int[] ageInfo = {12, 30, 45, 55};`
`int sumAge = 0;`
`for (int i = 0; i < ageInfo.length; i++)`
`  sumAge += ageInfo[i];`
	`int[] ageInfo = {12, 30, 45, 55};`
	`int sumAge = 0;`
	`for (int element : ageInfo)`
	`  sumAge += element;`

  - Note that an array element of a primitive value *cannot* be modified in the `for(:)` loop.

Iterating over non-generic Collections:

for(;;) Loop	for(:) Loop

`Collection nameList = new ArrayList();`
`nameList.add("Tom");`
`nameList.add("Dick");`
`nameList.add("Harry");`
`for (Iterator it = nameList.iterator();`
`  it.hasNext(); ) {`
`  Object element = it.next();`
`  if (element instanceof String) {`
`  String name = (String) element;`
`  //...`
`  }`
`}`
	`Collection nameList = new ArrayList();`
	`nameList.add("Tom");`
	`nameList.add("Dick");`
	`nameList.add("Harry");`
	`for (Object element : nameList) {`
	`  if (element instanceof String) {`
	`  String name = (String) element;`
	`  //...`
	`  }`
	`}`

Iterating over generic Collections:

for(;;) Loop	for(:) Loop

`Collection<String> nameList =`
`  new ArrayList<String>();`
`nameList.add("Tom");`
`nameList.add("Dick");`
`nameList.add("Harry");`
`for (Iterator<String> it = nameList.iterator();`
`  it.hasNext(); ) {`
`  String name = it.next();`
`  //...`
`}`
	`Collection<String> nameList =`
	`  new ArrayList<String>();`
	`nameList.add("Tom");`
	`nameList.add("Dick");`
	`nameList.add("Harry");`
	`for (String name : nameList) {`
	`  //...`
	`}`

- Note that syntax of the `for(:)` loop does not use an iterator for the collection.
- The `for(:)` loop does not allow elements to be removed from the collection.

The `for(:)` Loop

```
for (Type FormalParameter : Expression)
    Statement
```

- The *FormalParameter* must be declared in the `for(:)` loop.
- The *Expression* is evaluated only once.

STATIC IMPORT

Static Import

- Analogous to the *package import facility.*
- Static import allows accessible *static members* (static fields, static methods, static member classes and interfaces, enum classes) declared in a type to be imported.
- Once imported, the static member can be used by its *simple name*, and need not be qualified.
- Avoids use of *Constant Interface antipattern*, i.e. defining constants in interfaces.
- Import applies to the whole compilation unit.

Syntax:

```
// Static-import-on-demand: imports all static members
import static FullyQualifiedTypeName.*;

// Single-static-import: imports a specific static member
import static FullyQualifiedTypeName.StaticMemberName;
```

Note that import from the unnamed package (a.k.a. default package) is not permissible.

Avoiding the Constant Interface Antipattern

Constant Interface

```
package mypackage;
public interface MachineStates {
  // Fields are public,
  // static and final.
  int BUSY = 1;
  int IDLE = 0;
  int BLOCKED = -1;
}
```

Without Static Import

```
class MyFactory implements
                mypackage.MachineStates {
  public static void main(String[] args) {
    int[] states = {IDLE, BUSY, IDLE, BLOCKED };
    for (int s : states)
        System.out.println(s);
  }
}
```

With Static Import

```
import static mypackage.MachineStates.*; // Imports all static members.
class MyFactory2 {
  public static void main(String[] args) {
    int[] states = { IDLE, BUSY, IDLE, BLOCKED };
    for (int s : states)
        System.out.println(s);
  }
}
```

Static-import-on-demand: Import of All Static Members

Without Static Import	With Static Import

`class Calculate1 {`
`  public static void main(String[] args) {`
`  double x = 10.0, y = 20.5;`
`  double squareroot = Math.sqrt(x);`
`  double hypotenue = Math.hypot(x, y);`
`  double area = Math.PI * y * y;`
`  }`
`}`
	`import static java.lang.Math.*;`
	`// All static members from Math are imported.`
	`class Calculate2 {`
	`  public static void main(String[] args) {`
	`  double x = 10.0, y = 20.5;`
	`  double squareroot = sqrt(x);`
	`  double hypotenue = hypot(x, y);`
	`  double area = PI * y * y;`
	`  }`
	`}`

Single-static-import: Import of Individual Static Members

```
import static java.lang.Math.sqrt; // Static method
import static java.lang.Math.PI;   // Static field
// Only specified static members are imported.
class Calculate3 {
  public static void main(String[] args) {
    double x = 10.0, y = 20.5;
    double squareroot = sqrt(x);
    double hypotenue = Math.hypot(x, y); // Requires type name.
    double area = PI * y * y;
  }
}
```

Importing Enums

```
package mypackage;

public enum States { BUSY, IDLE, BLOCKED }
```

```
// Single type import
import mypackage.States;

// Static import from enum States
import static mypackage.States.*;

// Static import of static field
import static java.lang.System.out;

class Factory {
  public static void main(String[] args) {
    States[] states = {
         IDLE, BUSY, IDLE, BLOCKED
    };
    for (States s : states)
        out.println(s);
  }
}
```


FORMATTED OUTPUT

Formatted Output

- Classes `PrintStream` and `PrintWriter` provide the following convenience methods for formatting output:

`printf(String format, Object... args)` `printf(Locale l, String format, Object... args)`	Writes a formatted string using the specified *format string* and *argument list*.

- *Format string syntax* provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output.

Format String Syntax

- The format string can specify fixed text and embedded *format specifiers*.

```
System.out.printf("Formatted output|%6d|%8.3f|kr. |%.2f%n",
                                  2004, Math.PI, 1234.0354);
```

Output (Default locale Norwegian):

```
Formatted output|  2004|   3,142|kr. |1234,04|
```

  - The format string is the first argument.
  - It contains three format specifiers `%6d`, `%8.3f`, and `%.2f` which indicate *how* the arguments should be processed and *where* the arguments should be inserted in the format string.
  - All other text in the format string is fixed, including any other spaces or punctuation.
  - The argument list consists of all arguments passed to the method after the format string. In the above example, the argument list is of size three.
  - In the above example, the first argument is formatted according to the first format specifier, the second argument is formatted according to the second format specifier, and so on.

Format Specifiers for General, Character, and Numeric Types

%[argument_index`$`*][flags][width][*`.`*precision]conversion*

- The characters `%`, `$` and `.` have special meaning in the context of the format specifier.
- The optional *argument_index* is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "`1$`", the second by "`2$`", and so on.
- The optional *flags* is a set of characters that modify the output format. The set of valid flags depends on the conversion.
- The optional *width* is a decimal integer indicating the minimum number of characters to be written to the output.
- The optional *precision* is a decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.
- The required *conversion* is a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type.

Conversion Categories

General (`'b'`, `'B'`, `'h'`, `'H'`, `'s'`, `'S'`):

May be applied to any argument type.

Character (`'c'`, `'C'`):

May be applied to basic types which represent Unicode characters: `char`, `Character`, `byte`, `Byte`, `short`, and `Short`.

Numeric:

Integral (`'d'`, `'o'`, `'x'`, `'X'`):

May be applied to integral types: `byte`, `Byte`, `short`, `Short`, `int` and `Integer`, `long`, `Long`, and `BigInteger`.

Floating Point (`'e'`, `'E'`, `'f'`, `'g'`, `'G'`, `'a'`, `'A'`):

May be applied to floating-point types: `float`, `Float`, `double`, `Double`, and `BigDecimal`.

Percent (`'%'`): produces a literal `'%'`, i.e. "%%" escapes the `'%'` character.

Line Separator (`'n'`): produces the platform-specific line separator, i.e. "`%n`".

Conversion Table

- Upper-case conversions convert the result to upper-case according to the locale.

Conversion Specification	Conversion Category	**Description**
`'b','B'`	general	If the argument *arg* is `null`, then the result is `"false"`. If *arg* is a `boolean` or `Boolean`, then the result is string returned by `String.valueOf()`. Otherwise, the result is `"true"`.
`'h','H'`	general	If the argument *arg* is `null`, then the result is `"null"`. Otherwise, the result is obtained by invoking `Integer.toHexString(arg.hashCode())`.
`'s','S'`	general	If the argument *arg* is `null`, then the result is `"null"`. If *arg* implements `Formattable`, then `arg.formatTo()` is invoked. Otherwise, the result is obtained by invoking `arg.toString()`.
`'c','C'`	character	The result is a Unicode character.
`'d'`	integral	The result is formatted as a decimal integer.
`'o'`	integral	The result is formatted as an octal integer.
`'x','X'`	integral	The result is formatted as a hexadecimal integer.
`'e','E'`	floating point	The result is formatted as a decimal number in computerized scientific notation.
`'f'`	floating point	The result is formatted as a decimal number.
`'g','G'`	floating point	The result is formatted using computerized scientific notation for large exponents and decimal format for small exponents.

Conversion Specification	Conversion Category	**Description**
`'a','A'`	floating point	The result is formatted as a hexadecimal floating-point number with a significand and an exponent.
`'%'`	percent	The result is a literal `'%'` (`'\u0025'`).
`'n'`	line separator	The result is the platform-specific line separator.

Precision

- For general argument types, the precision is the maximum number of characters to be written to the output.
- For the floating-point conversions:

  If the conversion is `'e'`, `'E'` or `'f'`, then the precision is the number of digits after the decimal separator.

  If the conversion is `'g'` or `'G'`, then the precision is the total number of digits in the magnitude.

  If the conversion is `'a'` or `'A'`, then the precision must not be specified.
- For character, integral, and the percent and line separator conversions:
  - The precision is not applicable.
  - If a precision is provided, an exception will be thrown.

Flags

- *y* means the flag is supported for the indicated argument types.

Flag	General	Character	Integral	Floating Point	Date/Time	Description
`'-'`	y	y	y	y	y	The result will be left-justified.
`'#'`	y[1]	-	y[3]	y	-	The result should use a conversion-dependent alternate form.
`'+'`	-	-	y[4]	y	-	The result will always include a sign.
`' '`	-	-	y[4]	y	-	The result will include a leading space for positive values.
`'0'`	-	-	y	y	-	The result will be zero-padded.
`','`	-	-	y[2]	y[5]	-	The result will include locale-specific grouping separators.
`'('`	-	-	y[4]	y[5]	-	The result will enclose negative numbers in parentheses.

[1] Depends on the definition of `Formattable`.

[2] For `'d'` conversion only.

[3] For `'o'`, `'x'` and `'X'` conversions only.

[4] For `'d'`, `'o'`, `'x'` and `'X'` conversions applied to `BigInteger` or `'d'` applied to `byte`, `Byte`, `short`, `Short`, `int` , `Integer`, `long`, and `Long`.

[5] For `'e'`, `'E'`, `'g'` and `'G'` conversions only.

Examples: Formatted Output

```
// Argument Index: 1$, 2$, ...
String fmtYMD = "Year-Month-Day: %3$s-%2$s-%1$s%n";
String fmtDMY = "Day-Month-Year: %1$s-%2$s-%3$s%n";
out.printf(fmtYMD, 7, "March", 2004);
out.printf(fmtDMY, 7, "March", 2004);

// General ('b', 'h', 's')
out.printf("1|%b|%b|%b|%n", null, true, "BlaBla");
out.printf("2|%h|%h|%h|%n", null, 2004, "BlaBla");
out.printf("3|%s|%s|%s|%n", null, 2004, "BlaBla");
out.printf("4|%.1s|%.2s|%.3s|%n",    null, 2004, "BlaBla");
out.printf("5|%6.1s|%4.2s|%2.3s|%n", null, 2004, "BlaBla");
out.printf("6|%2$s|%3$s|%1$s|%n",    null, 2004, "BlaBla");
out.printf("7|%2$4.2s|%3$2.3s|%1$6.1s|%n",   null, 2004, "BlaBla");
```

```
Year-Month-Day: 2004-March-7
Day-Month-Year: 7-March-2004

1|false|true|true|
2|null|7d4|76bee0a0|
3|null|2004|BlaBla|
4|n|20|Bla|
5|     n|  20|Bla|
6|2004|BlaBla|null|
7|  20|Bla|     n|
```


Examples: Formatted Output (cont.)

```
// Integral ('d', 'o', 'x')
out.printf("1|%d|%o|%x|%n", (byte)63, 63, (Long)63L);
out.printf("2|%d|%o|%x|%n",
           (byte)-63, -63, (Long)(-63L));
out.printf("3|%+05d|%-+5d|%+d|%n", -63, 63, 63);
out.printf("4|% d|% d|%(d|%n", -63, 63, -63);
out.printf("5|%-, 10d|%, 10d|%,(010d|%n",
            -654321, 654321, -654321);

// Floating Point ('e', 'f', 'g', 'a')
out.printf("1|%e|%f|%g|%a|%n", E, E, E, E);

out.printf("3|%-, 12.3f|%, 12.2f|%,(012.1f|%n",
            -E*1000.0, E*1000.0, -E*1000.0);

for(int i = 0; i < 4; ++i) {
  for(int j = 0; j < 3; ++j)
    out.printf("%,10.2f", random()*10000.0);
  out.println();
}
```

```
1|63|77|3f|
2|-63|37777777701|ffffffffffffffc1|

3|-0063|+63  |+63|
4|-63| 63|(63)|
5|-654 321  |   654 321|(0654 321)|

1|2.718282e+00|2,718282|2,718282|0x1.5bf0a8b1
45769p1|
3|-2 718,282  |    2 718,28|(0002 718,3)|

  2 449,24  1 384,41  2 826,67
  1 748,13  8 904,55  9 583,94
  4 261,83  9 203,00  1 753,60
  7 315,30  8 918,63  2 936,91
```

FORMATTED INPUT

Formatted Input

- Class `java.util.Scanner` implements a simple text scanner (*lexical analyzer*) which uses *regular expressions* to parse primitive types and strings from its source.
- A Scanner converts the input from its source into *tokens* using a *delimiter pattern*, which by default matches *whitespace*.
- The tokens can be converted into values of different types using the various `next()` methods.

```
Scanner lexer1 = new Scanner(System.in); // Connected to standard input.
int i = lexer1.nextInt();
...
Scanner lexer2 = new Scanner(new File("myLongNumbers")); (1) Construct a scanner.
while (lexer2.hasNextLong()) {    // (2) End of input? May block.
  long aLong = lexer2.nextLong(); // (3) Deal with the current token. May block.
}
lexer2.close();                   // (4) Closes the scanner. May close the source.
```

- Before parsing the next token with a particular `next()` method, for example at (3), a *lookahead* can be performed by the corresponding `hasNext()` method as shown at (2).
- 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.

`java.util.Scanner` Class API

- Constructing a Scanner
- Lookahead Methods
- Parsing the Next Token
- Misc. Scanner Methods

Constructing a Scanner

- A scanner must be constructed to parse text.

`Scanner(`*`Type`* `source)`
Returns an appropriate scanner. *Type* can be a `String`, a `File`, an `InputStream`, a `ReadableByteChannel`, or a `Readable` (implemented by `CharBuffer` and various `Readers`).

Scanning

- A scanner throws an `InputMismatchException` when it cannot parse the input.

Lookahead Methods	Parsing the Next Token
`boolean hasNext()` The method returns `true` if this scanner has another token in its input.	`String next()` The method scans and returns the next complete token from this scanner.

`boolean hasNext`*IntegralType*`()`
`boolean hasNext`*IntegralType*`(int radix)`
Returns `true` if the next token in this scanner's input can be interpreted as an *IntegralType*`'` value corresponding to *IntegralType* in the default or specified radix.
	IntegralType`' next`*IntegralType*`()`
	IntegralType`' next`*IntegralType*`(int radix)`
	Scans the next token of the input as a *IntegralType*`'` value corresponding to *IntegralType*.

The name *IntegralType* can be `Byte`, `Short`, `Int`, `Long`, or `BigInteger`. The corresponding *IntegralType*`'` can be `byte`, `short`, `int`, `long` or `BigInteger`.	
`boolean hasNext`*FPType*`()` Returns `true` if the next token in this scanner's input can be interpreted as a *FPType*`'` value corresponding to *FPType*.	*FPType*`' next`*FPType*`()` Scans the next token of the input as a *FPType*`'` value corresponding to *FPType*.
The name *FPType* can be `Float`, `Double` or `BigDecimal`. The corresponding *FPType*`'` can be `float`, `double` and `BigDecimal`.	

Lookahead Methods	Parsing the Next Token
`boolean hasNextBoolean()` Returns `true` if the next token in this scanner's input can be interpreted as a `boolean` value using a case insensitive pattern created from the string `"true|false"`.	`boolean nextBoolean()` Scans the next token of the input into a `boolean` value and returns that value.
	`String nextLine()` Advances this scanner past the current line and returns the input that was skipped.

Misc. Scanner Methods

`void close()`

Closes this scanner. When a scanner is closed, it will close its input source if the source implements the `Closeable` interface (implemented by various `Channels`, `InputStreams`, `Readers`).

`Pattern delimiter()`

Returns the pattern this scanner is currently using to match delimiters.

`Scanner useDelimiter(Pattern pattern)`

Sets this scanner's delimiting pattern to the specified pattern.

`Scanner useDelimiter(String pattern)`

Sets this scanner's delimiting pattern to a pattern constructed from the specified `String`.

`int radix()`

Returns this scanner's default radix.

`Scanner useRadix(int radix)`

Sets this scanner's default radix to the specified radix.

`Locale locale()`

Returns this scanner's locale.

`Scanner useLocale(Locale locale)`

Sets this scanner's locale to the specified locale.

Examples: Reading from the Console

```
/* Reading from the console. */
import java.util.Scanner;

import static java.lang.System.out;

public class ConsoleInput {

  public static void main(String[] args) {

    // Create a Scanner which is chained to System.in, i.e. to the console.
    Scanner lexer = new Scanner(System.in);

    // Read a list of integers.
    int[] intArray = new int[3];
    out.println("Input a list of integers (max. " + intArray.length + "):");
    for (int i = 0; i < intArray.length;i++)
      intArray[i] = lexer.nextInt();
    for (int i : intArray)
       out.println(i);
```

```
    // Read names
    String firstName;
    String lastName;
    String name;
    String repeat;
    do {
      lexer.nextLine(); // Empty any input still in the current line
      System.out.print("Enter first name: ");
      firstName = lexer.next();
      lexer.nextLine();
      System.out.print("Enter last name: ");
      lastName = lexer.next();
      lexer.nextLine();
      name = lastName + " " + firstName;
      System.out.println("The name is " + name);
      System.out.print("Do Another? (y/n): ");
      repeat = lexer.next();
    } while (repeat.equals("y"));
    lexer.close();
  }
}
```


Dialogue with the program:

```
Input a list of integers (max. 3):
23 45 55
23
45
55
Enter first name: Java
Enter last name: Jive
The name is Jive Java
Continue? (y/n): y
Enter first name: Sunny Java
Enter last name: Bali
The name is Bali Sunny
Continue? (y/n): n
```

Dialogue with the program:

```
Input a list of integers (max. 3):
23 4..5 34
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at ConsoleInput.main(ConsoleInput.java:17)
```

Examples: Using a Scanner

```
{
  String input = "The world will end today.";
  Scanner lexer = new Scanner(input);
  while (lexer.hasNext())
    out.println(lexer.next());
  lexer.close();
}
```

```
The
world
will
end
today.
```

```
{
  String input = "123 45,56 false 567 722 blabla";
    Scanner lexer = new Scanner(input);
    out.println(lexer.hasNextInt());
    out.println(lexer.nextInt());
    out.println(lexer.hasNextDouble());
    out.println(lexer.nextDouble());
    out.println(lexer.hasNextBoolean());
    out.println(lexer.nextBoolean());
    out.println(lexer.hasNextInt());
    out.println(lexer.nextInt());
    out.println(lexer.hasNextLong());
    out.println(lexer.nextLong());
    out.println(lexer.hasNext());
    out.println(lexer.next());
    out.println(lexer.hasNext());
    lexer.close();
}
```

```
true
123
true
45.56
true
false
true
567
true
722
true
blabla
false
```


Examples: Using Delimiters with a Scanner

```
{  // Using default delimiters (i.e. whitespace).
   // Note local locale format for floating-point numbers.
   String input = "123 45,56 false 567 722 blabla";
   String delimiters = "default";
   parse(input, delimiters, INT, DOUBLE, BOOL, INT, LONG, STR);
}
```

```
Input: "123 45,56 false 567 722
blabla"
Delimiters: (default)123
45.56
false
567
722
blabla
```

```
{  // Note the use of backslash to escape characters in regexp.
   String input = "2004 | 2 | true";
   String delimiters = "\\s*\\|\\s*";
   parse(input, delimiters, INT, INT, BOOL);
}
```

```
Input: "2004 | 2 | true"
Delimiters: (\s*\|\s*)
2004
2
true
```

```
{  // Another example of a regexp to specify delimiters.
   String input = "Always = true | 2 $ U";
   String delimiters = "\\s*(\\||\\$|=)\\s*";
   parse(input, delimiters, STR, BOOL, INT, STR);
}
```

```
Input: "Always = true | 2 $ U"
Delimiters: (\s*(\||\$|=)\s*)
Always
true
2
U
```

```
/** Parses the input using the delimiters and expected sequence of tokens. */
public static void parse(String input, String delimiters, TOKEN_TYPE... sequence) {
  out.println("Input: \"" + input + "\"");
  out.println("Delimiters: (" + delimiters + ")");

  Scanner lexer = new Scanner(input);           // Construct a scanner.
  if (!delimiters.equalsIgnoreCase("default"))  // Set delimiters if necessary.
    lexer.useDelimiter(delimiters);

  for (TOKEN_TYPE tType : sequence) {      // Iterate through the tokens.
    if (!lexer.hasNext()) break;           // Handle premature end of input.
    switch(tType) {
      case INT:    out.println(lexer.nextInt()); break;
      case LONG:   out.println(lexer.nextLong()); break;
      case FLOAT:  out.println(lexer.nextFloat()); break;
      case DOUBLE: out.println(lexer.nextDouble()); break;
      case BOOL:   out.println(lexer.nextBoolean()); break;
      case STR:    out.println(lexer.next()); break;
      default:     assert false;
    }
  }
  lexer.close();                           // Close the scanner.
}
```