Java Jive from 1.4 to 5

(Extended Version)

Advanced Topics in Java

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

Version date: 2006-01-17

Overview

• Enumerated types • Automatic Boxing and Unboxing of Primitive Values • Enhanced for loop	• Static Import • Varargs • Formatted Output • Formatted Input
Topics not covered here: • Generic Types • Concurrency • Metadata	

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 Pattern

- Details in "*Effective Java Programming Language Guide*", ISBN 0201310058.
- A class exports *self-typed constants* (see (1), (2) and (3)).
- Fixes disadvantages of the `int` enum pattern, but it is verbose and error prone.

```
import java.util.*;
import java.io.*;

// Class can implement interfaces.
public class MachineState implements Comparable, Serializable {

  // Descriptive name of the constant
  private final String name;
  public String toString() { return name; }

  // Private constructor which cannot be called to create more objects.
  private MachineState(String name) { this.name = name; }

  // Self-typed constants
  public static final MachineState BUSY = new MachineState("Busy");       // (1)
  public static final MachineState IDLE = new MachineState("Idle");       // (2)
  public static final MachineState BLOCKED = new MachineState("Blocked"); // (3)
```

```
// Constants are assigned values consecutively, starting with 0.
// Static counter to keep track of values assigned so far.
private static int nextOrdinal = 0;

// The value in a constant is the current value of nextOrdinal.
private final int ordinal = nextOrdinal++;

// Prevent overriding of equals() from Object class.
public final boolean equals(Object o) {
  return super.equals(o);
}

// Prevent overriding of hashode() from Object class.
public final int hashCode() {
  return super.hashCode();
}

// Natural order of constants is declaration order.
public int compareTo(Object o) {
  return ordinal - ((MachineState)o).ordinal;
}
```



```
  // Used in serialization
  private static final MachineState[] PRIVATE_VALUES =
                 { BUSY, IDLE, BLOCKED };
  public static final List VALUES = Collections.unmodifiableList(
                  Arrays.asList(PRIVATE_VALUES));
  private Object readResolve() {
    // Canonicalize
    return PRIVATE_VALUES[ordinal];
  }
  // Other members can be added.
}
```

- Using the typesafe enum pattern:

```
public class Machine {
  MachineState state;
  public void
  setState(MachineState state) {
    this.state = state;
  }
  public MachineState getState() {
    return this.state;
  }
}
```

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

    Machine machine = new Machine();
    machine.setState(MachineState.BUSY);
    // machine.setState(1); // Compile error
    System.out.println(MachineState.BUSY);
    // Prints "Busy"
  }
}
```

Comments to the Typesafe Enum Pattern

- No new objects can be created -- only objects denoted by the self-typed fields exist.
- Class can be augmented, extended, serialized and used in collections.
- Behaviour of the constants can be tailored by nested classes, for example by using anonymous classes to implement the constants.
- Disadvantages:
  – Not very efficient to aggregate typesafe enum constants into sets.
  – Cannot be used as `case` labels in a `switch` statement.

Typesafe Enum Construct

- The enum construct provides support for the typesafe enum pattern:

```
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*.
- Used in the same way as the typesafe enum pattern.
- Has all the advantages of the typesafe enum pattern, but is more powerful!
- No changes in the JVM.

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.
  - It is implicitly static, i.e. no outer object is associated with an enum constant.
  - It is implicitly `final` unless it contains *constant-specific class bodies* (see below), but it can implement interfaces.
  - It cannot be declared `abstract` unless each abstract method is overridden in the constant-specific class body of every enum constant (see below).

```
// (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;
    }
  }
}
```


Extending Enum Types: Constant-Specific Class Bodies

- Constant-specific class bodies define anonymous classes inside an enum type that extend the enclosing enum type.
- Instance methods declared in these class bodies are accessible outside the enclosing enum type only if they override accessible methods in the enclosing enum type.
- An enum type that contains constant-specific class bodies cannot be declared `final`.
- The enum type `Meal` below illustrates declaration of constant-specific class bodies.
  - The abstract method `mealPrice()` is overridden in each constant-specific body.
  - Given that references `meal` and `day` are of enum types `Meal` and `Day` respectively, the method call
    `meal.mealPrice(day)`
    will execute the `mealPrice()` method from the constant-specific body of the enum constant denoted by the reference `meal`, as would be expected.
  - See the following source code files for usage of the `Meal` enum type:
    `DayMeal2.java`, `DayMealPlanner2.java`.

```
public enum Meal {
  // Each enum constant defines a constant-specific class body
  BREAKFAST(7,30) {
    public double mealPrice(Day day) {
      double breakfastPrice = 10.50;
      if (weekend.contains(day))
          breakfastPrice *= 1.5;
      return breakfastPrice;
    }
  },
  LUNCH(12,15) {
    public double mealPrice(Day day) {
      double lunchPrice = 20.50;
      if (weekend.contains(day))
          lunchPrice *= 2.0;
      return lunchPrice;
    }
  },
  DINNER(19,45) {
    public double mealPrice(Day day) {
      double dinnerPrice = 25.50;
      if (weekend.contains(day))
          dinnerPrice *= 2.5;
```



```
      return dinnerPrice;
    }
  };
  // Abstract method which the constant-specific class body
  abstract double mealPrice(Day day);
  // Static field referenced in the constant-specific class bodies.
  private static final EnumSet<Day> weekend =
                   EnumSet.range(Day.SATURDAY, Day.SUNDAY);
  // Enum constructor
  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;
  }
  // Instance fields: Time for the meal.
  private int hh;
  private int mm;
  // Instance methods
  public int getHour() { return this.hh; }
  public int getMins() { return this.mm; }

} // End Meal
```

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.
- It is illegal to reference a static field of an enum type from its constructors or instance initializers.
  - See the following source code file: `DayOfTheWeek.java`.

Extending the Java Collections Framework: `EnumSet`

- Enums can be used in a special purpose `Set` implementation (`EnumSet`) which provides better performance.
- All of the elements in an enum set must come from a single enum type.
- Enum sets are represented internally as bit vectors.
- The `EnumSet` class defines new methods and inherits the ones in the `Set/Collection` interfaces (see below).
- All methods are `static` except for the `clone()` method.
- All methods return an `EnumSet<E>`.
- The `null` reference cannot be stored in an enum set.

Method Summary for the abstract class `EnumSet<E extends Enum<E>>`
`allOf(Class<E> elementType)` Creates an enum set containing all of the elements in the specified element type.
`clone()` Returns a copy of this set.
`complementOf(EnumSet<E> s)` Creates an enum set with the same element type as the specified enum set, initially containing all the elements of this type that are not contained in the specified set.

Method Summary for the abstract class `EnumSet<E extends Enum<E>>`
`copyOf(Collection<E> c)` Creates an enum set initialized from the specified collection.
`copyOf(EnumSet<E> s)` Creates an enum set with the same element type as the specified enum set, initially containing the same elements (if any).
`noneOf(Class<E> elementType)` Creates an empty enum set with the specified element type.

`of(E e)`
`of(E e1, E e2)`
`of(E e1, E e2, E e3)`
`of(E e1, E e2, E e3, E e4)`
`of(E e1, E e2, E e3, E e4, E e5)`
These factory methods creates an enum set initially containing the specified element(s). Note that enums cannot be used in varargs as it is not legal to use varargs with parameterized types.

`range(E from, E to)` Creates an enum set initially containing all of the elements in the range defined by the two specified endpoints.


```
import java.util.EnumSet;
public class UsingEnumSet {
  public static void main(String[] args) {
    EnumSet<Day> allDays = EnumSet.range(Day.MONDAY, Day.SUNDAY);
    System.out.println("All days: " + allDays);
    EnumSet<Day> weekend = EnumSet.range(Day.SATURDAY, Day.SUNDAY);
    System.out.println("Weekend: " + weekend);

    EnumSet<Day> oddDays = EnumSet.of(Day.MONDAY, Day.WEDNESDAY,
                                      Day.FRIDAY, Day.SUNDAY);
    System.out.println("Odd days: " + oddDays);

    EnumSet<Day> evenDays = EnumSet.complementOf(oddDays);
    System.out.println("Even days: " + evenDays);

    EnumSet<Day> weekDays = EnumSet.complementOf(weekend);
    System.out.println("Week days: " + weekDays);
  }
}
```

Output from the program:

```
All days: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]
Weekend: [SATURDAY, SUNDAY]
Odd days: [MONDAY, WEDNESDAY, FRIDAY, SUNDAY]
Even days: [TUESDAY, THURSDAY, SATURDAY]
Week days: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY]
```

Extending the Java Collections Framework: EnumMap

- Enums can be used in a special purpose Map implementation (EnumMap) which provides better performance.
- Enum maps are represented internally as arrays.
- All of the *keys* in an enum map must come from a single enum type.
- Enum maps are maintained in the *natural order of their keys* (i.e. the order of the enum constant declarations).
- The EnumMap class re-implements most of the methods in the Map interface (see below).
- The null reference as a key is not permitted.

Constructor Summary of the class EnumMap<K extends Enum<K>,V>
EnumMap(Class<K> keyType) Creates an empty enum map with the specified key type.
EnumMap(EnumMap<K,? extends V> m) Creates an enum map with the same key type as the specified enum map, initially containing the same mappings (if any).
EnumMap(Map<K,? extends V> m) Creates an enum map initialized from the specified map.

Method Summary of the class EnumMap<K extends Enum<K>,V>	
void	clear() Removes all mappings from this map.
EnumMap<K,V>	clone() Returns a shallow copy of this enum map.
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
boolean	equals(Object o) Compares the specified object with this map for equality.
V	get(Object key) Returns the value to which this map maps the specified key, or null if this map contains no mapping for the specified key.

Method Summary of the class EnumMap<K extends Enum<K>,V>

`Set<K>`	`keySet()` Returns a Set view of the keys contained in this map.
`V`	`put(K key, V value)` Associates the specified value with the specified key in this map.
`void`	`putAll(Map<? extends K,? extends V> m)` Copies all of the mappings from the specified map to this map.
`V`	`remove(Object key)` Removes the mapping for this key from this map if present.
`int`	`size()` Returns the number of key-value mappings in this map.
`Collection<V>`	`values()` Returns a `Collection` view of the values contained in this map.

Example of using EnumMap

```
import java.util.*;

public class UsingEnumMap {
  enum Day {
      MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
  }
  public static void main(String[] args) {


    int[] freqArray = {12, 34, 56, 23, 5, 13, 78};

    // Create a Map of frequencies
    Map<Day, Integer> ordinaryMap = new HashMap<Day, Integer>();
    for (Day day : Day.values())
        ordinaryMap.put(day, freqArray[day.ordinal()]);
    System.out.println("Frequency Map: " + ordinaryMap);

    // Create an EnumMap of frequencies
    EnumMap<Day, Integer> frequencyEnumMap =
                      new EnumMap<Day, Integer>(ordinaryMap);
```

```
    // Change some frequencies
    frequencyEnumMap.put(Day.MONDAY, 100);
    frequencyEnumMap.put(Day.FRIDAY, 123);
    System.out.println("Frequency EnumMap: " + frequencyEnumMap);

    // Values
    Collection<Integer> frequencies = frequencyEnumMap.values();
    System.out.println("Frequencies: " + frequencies);

    // Keys
    Set<Day> days = frequencyEnumMap.keySet();
    System.out.println("Days: " + days);
  }
}
```

Output from the program:

```
Frequency Map: {TUESDAY=34, THURSDAY=23, WEDNESDAY=56, MONDAY=12, SUNDAY=78,
SATURDAY=13, FRIDAY=5}
Frequency EnumMap: {MONDAY=100, TUESDAY=34, WEDNESDAY=56, THURSDAY=23,
FRIDAY=123, SATURDAY=13, SUNDAY=78}
Frequencies: [100, 34, 56, 23, 123, 13, 78]
Days: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]
```


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:

  If `p` is a value of a *primtiveType*, boxing conversion converts `p` into a reference `r` of corresponding *WrapperType*, such that `r.`*primitiveType*`Value() == p`.
- *Unboxing conversion* converts objects of wrapper types to values of corresponding primitive types:

  If `r` is a reference of a *WrapperType*, unboxing conversion converts the reference `r` into `r.`*primitiveType*`Value()`, there *primitiveType* is the primitive type corresponding to the *WrapperType*.

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
```

- Boxing and unboxing in collections/maps.

```
import static java.lang.System.out;

public static void main(String... args) {
    Map<String, Integer> freqMap = new TreeMap<String, Integer>();
    for (String word : args) {
        Integer freq = freqMap.get(word);
        freqMap.put(word, freq == null ? 1 : freq + 1);
    }
    out.println(freqMap);
}
```


ENHANCED FOR LOOP

Enhanced for Loop: for(:)

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

Iterating over Arrays:

<table>
<tr><th>for(;;) Loop</th><th>for(:) Loop</th></tr>
<tr><td><pre>int[] ageInfo = {12, 30, 45, 55};
int sumAge = 0;
for (int i = 0; i < ageInfo.length; i++)
    sumAge += ageInfo[i];</pre></td><td><pre>int[] ageInfo = {12, 30, 45, 55};
int sumAge = 0;
for (int element : ageInfo)
    sumAge += element;</pre></td></tr>
</table>

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

Iterating over non-generic Collections:

<table>
<tr><th>for(;;) Loop</th><th>for(:) Loop</th></tr>
<tr><td><pre>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;
        //...
    }
}</pre></td><td><pre>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;
        //...
    }
}</pre></td></tr>
</table>

Iterating over generic Collections:

<table>
<tr><th>for(;;) Loop</th><th>for(:) Loop</th></tr>
<tr><td><pre>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();
     //...
}</pre></td><td><pre>Collection<String> nameList =
          new ArrayList<String>();
nameList.add("Tom");
nameList.add("Dick");
nameList.add("Harry");
for (String name : nameList) {
     //...
}</pre></td></tr>
</table>

- 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.
- The *type* of *Expression* is `java.lang.Iterable` or an array.
  – The `java.util.Collection` interface is retrofitted to extend the `java.lang.Iterable` interface which has the method prototype `Iterator<T> iterator()`.
- When *Expression* is an array:
  – The *type* of the array element must be assignable to the *Type* of the *FormalParameter*.
- When *Expression* is an instance of a *raw type* `java.lang.Iterable`:
  – The *Type* of the *FormalParameter* must be `Object`.
- When *Expression* is an instance of a *parameterized type* `java.lang.Iterable<T>`:
  – The *type parameter* `T` must be assignable to the *Type* of the *FormalParameter*.
- The *labelled* `for(:)` loop has equivalent semantics to the *labelled* `for(;;)` loop as regards the use of `break` and `continue` statements in its body.
- A `for(:)` loop is transformed to a `for(;;)` loop at compile time.

Nested for(:) Loop

```
import java.util.*;
public class PlayingCards {
    enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
    enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN,
                EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }
    static class Card {
        Suit suit;
        Rank rank;
        Card(Suit s, Rank r) { suit = s; rank = r; }
        public String toString() { return "(" + suit + "," + rank + ")"; }
    }
    public static void main(String[] args) {
        List<Card> deck = new ArrayList<Card>();
        for (Suit suit : Suit.values())
            for (Rank rank : Rank.values())
                deck.add(new Card(suit, rank));
        System.out.println(deck);
        Collections.shuffle(deck);
        System.out.println(deck);
    }
}
```


Restrictions on the Enhanced for Loop

- Cannot be used with an `Iterator`.
- Cannot be used to remove elements from a collection.
- Cannot be used to modify the current slot in a collection or an array.
- Cannot be used to simultaneously iterate over multiple collections or arrays.

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	Without Static Import

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

Importing nested types

- Class `Machine` has 3 nested types.

```
package yap;

public class Machine {

  public static class StateContants {
    public static final String BUSY = "Busy";
    public static final String IDLE = "Idle";
    public static final String BLOCKED = "Blocked";
  }

  public enum StateEnum {
    BUSY, IDLE, BLOCKED
  }

  public enum StateEnumII {
    UNDER_REPAIR, WRITE_OFF, HIRED, AVAILABLE;
  }
}
```


Importing nested types (cont.)

- The effect of the following import declarations is shown in the code for the `MachineClient`.

```
import yap.Machine;                             // (0)

import yap.Machine.StateContants;               // (1)
import static yap.Machine.StateContants;        // (2) Superfluous because of (1)
import static yap.Machine.StateContants.*;      // (3)

import yap.Machine.StateEnum;               // (4)
import static yap.Machine.StateEnum;        // (5) Superfluous because of (4)
import static yap.Machine.StateEnum.*;      // (6)

import yap.Machine.StateEnumII;             // (7)
import static yap.Machine.StateEnumII;      // (8) Superfluous because of (7)
import static yap.Machine.StateEnumII.*;    // (9)
import static yap.Machine.StateEnumII.WRITE_OFF;  // (10)
```

- Code illustrates various scenarios using import declarations.

```
public class MachineClient {
  public static void main(String[] args) {
    StateContants msc = new StateContants(); // Requires (1) or (2)
//  String intState = IDLE;                  // Ambiguous because of (3)and (6)
    String intState = StateContants.IDLE;    // Explicit disambiguation required

//  StateEnum se = BLOCKED;                // Ambiguous because of (3)and (6)
    StateEnum se = StateEnum.BLOCKED;      // Requires (4) or (5)
    StateEnum enumState = StateEnum.IDLE;  // Explicit disambiguation required

    StateEnumII[] states = {               // Requires (7) or (8)
        AVAILABLE, HIRED, UNDER_REPAIR,    // Requires (9)
        WRITE_OFF,                         // Requires (9) or (10)
        StateEnumII.WRITE_OFF,             // Requires (7) or (8)
        Machine.StateEnumII.WRITE_OFF,     // Requires (0)
        yap.Machine.StateEnumII.WRITE_OFF  // Does not require any import
    };
    for (StateEnumII s : states)
        System.out.println(s);
  }
}
```


VARARGS

Varargs: Variable-Arity Methods

- Purpose: add methods that can be called with variable-length argument list.
- Heavily employed in formatting text output, aiding internationalization.
- Syntax and semantics:
  - the *last* formal parameter in a method declaration can be declared as: *ReferenceType*`...` *FormalParameterName*
  - the *last* formal parameter in the method is then interpreted as having the type: *ReferenceType*`[]`

```
// Method declaration
public static void publish(String str, Object... data) // Object[]
  { ... }
```

  - Given that the method declaration has *n* formal parameters and the method call has *k* actual parameters, then *k* must be equal or greater than *n-1*.
  - The last *k-n+1* actual parameters are evaluated and stored in an array whose reference value is passed as the actual parameter.

```
// Method calls
publish("one");           // ("one", new Object[] {})
publish("one", "two");    // ("one", new Object[] {"two"})
publish("one", "two", 3); // ("one", new Object[] {"two", new Integer(3)})
```


Example: Varargs

- Variable-arity method calls to a variable-arity method.

```
import static java.lang.System.out;
class VarargsDemo {
  public static void main(String... args) { // main() as varargs method
    out.println(args.length + " arguments:");
    for(String str : args)
      out.println(str);

    int    day   = 1;
    String month = "March";
    int    year  = 2004;

    flexiPrint();                 // new Object[] {}
    flexiPrint(day);              // new Object[] {new Integer(day)}
    flexiPrint(day, month);       // new Object[] {new Integer(day),
                                  //               month}
    flexiPrint(day, month, year); // new Object[] {new Integer(day),
                                  //               month,
                                  //               new Integer(year)}
  }
```

```
  public static void flexiPrint(Object... data) { // Object[]
    out.println("No. of elements: " + data.length);
    for(int i = 0; i < data.length; i++)
      out.print(data[i] + " ");
    out.println();
  }
}
```

Running the program:

```
>java -ea VarargsDemo Java 1.5 rocks!
3 arguments:
Java
1.5
rocks!
No. of elements: 0

No. of elements: 1
1
No. of elements: 2
1 March
No. of elements: 3
1 March 2004
```


Overloading Resolution

- Resolution of overloaded methods selects the *most specific* method for execution.
- One method is more specific than another method if all actual parameters that can be accepted by the one can be accepted by the other.
  - A method call can lead to an ambiguity between two or more overloaded methods, and is flagged by the compiler.

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

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

Varargs and Overloading

- The example illustrates how the most specific overloaded method is chosen for a method call.

```
import static java.lang.System.out;
class VarargsOverloading {
  public void operation(String str) {
    String signature = "(String)";
    out.println(str + " => " + signature);
  }
  public void operation(String str, int m) {
    String signature = "(String, int)";
    out.println(str + " => " + signature);
  }
  public void operation(String str, int m, int n) {
    String signature = "(String, int, int)";
    out.println(str + " => " + signature);
  }
  public void operation(String str, Integer... data) {
    String signature = "(String, Integer[])";
    out.println(str + " => " + signature);
  }
```



```
  public void operation(String str, Number... data) {
    String signature = "(String, Number[])";
    out.println(str + " => " + signature);
  }
  public void operation(String str, Object... data) {
    String signature = "(String, Object[])";
    out.println(str + " => " + signature);
  }
  public static void main(String[] args) {
    VarargsOverloading ref = new VarargsOverloading();
    ref.operation("(String)");
    ref.operation("(String, int)",           10);
    ref.operation("(String, Integer)",       new Integer(10));
    ref.operation("(String, int, byte)",     10, (byte)20);
    ref.operation("(String, int, int)",      10,  20);
    ref.operation("(String, int, long)",     10,  20L);
    ref.operation("(String, int, int)",      10,  20,  30);
    ref.operation("(String, int, double)",   10,  20.0);
    ref.operation("(String, int, String)",   10,  "what?");
    ref.operation("(String, boolean)",       false);
  }
}
```

Output from the program:

```
(String) => (String)
(String, int) => (String, int)
(String, Integer) => (String, int)
(String, int, byte) => (String, int, int)
(String, int, int) => (String, int, int)
(String, int, long) => (String, Number[])
(String, int, int) => (String, Integer[])
(String, int, double) => (String, Number[])
(String, int, String) => (String, Object[])
(String, boolean) => (String, Object[])
```


Varargs and Overriding

- Overriding of varargs methods does not present any surprises as along as criteria for overriding is satisfied.

```
import static java.lang.System.out;
public class OneSuperclass {
  public int doIt(String str, Integer... data)
             throws java.io.EOFException, java.io.FileNotFoundException { // (1)
    String signature = "(String, Integer[])";
    out.println(str + " => " + signature);
    return 1;
  }
  public void doIt(String str, Number... data) {  // (2)
    String signature = "(String, Number[])";
    out.println(str + " => " + signature);
  }
}
---------------------------------------------------------------------------
```

```
import static java.lang.System.out;

public class OneSubclass extends OneSuperclass {

//public int doIt(String str, Integer[] data)    // (1) Compile time error
  public int doIt(String str, Integer... data)   // Overriding
             throws java.io.FileNotFoundException {
    String signature = "(String, Integer[])";
    out.println("Overridden: " + str + " => " + signature);
    return 0;
  }

  public void doIt(String str, Object... data) { // Overloading
    String signature = "(String, Object[])";
    out.println(str + " => " + signature);
  }

  public static void main(String[] args) throws Exception {
    OneSubclass ref = new OneSubclass();
    ref.doIt("(String)");
    ref.doIt("(String, int)",           10);
    ref.doIt("(String, Integer)",       new Integer(10));
    ref.doIt("(String, int, byte)",     10, (byte)20);
```



```
    ref.doIt("(String, int, int)",      10,  20);
    ref.doIt("(String, int, long)",     10,  20L);
    ref.doIt("(String, int, int)",      10,  20,  30);
    ref.doIt("(String, int, double)",   10,  20.0);
    ref.doIt("(String, int, String)",   10,  "what?");
    ref.doIt("(String, boolean)",       false);
  }
}
```

Output from the program:

```
Overridden: (String) => (String, Integer[])
Overridden: (String, int) => (String, Integer[])
Overridden: (String, Integer) => (String, Integer[])
(String, int, byte) => (String, Number[])
Overridden: (String, int, int) => (String, Integer[])
(String, int, long) => (String, Number[])
Overridden: (String, int, int) => (String, Integer[])
(String, int, double) => (String, Number[])
(String, int, String) => (String, Object[])
(String, boolean) => (String, Object[])
```

FORMATTED OUTPUT

Formatted Output

[Coverage here is based on the Java SDK 1.5 API Documentation.]

- Classes `java.lang.String`, `java.io.PrintStream`, `java.io.PrintWriter` and `java.util.Formatter` provide the following overloaded methods for formatted output:

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

  - The `format()` method returns a `String`, a `PrintStream`, a `PrintWriter` or a `Formatter` respectively for these classes, allowing method call chaining.
  - The `format()` method is `static` in the `String` class.
- In addition, classes `PrintStream` and `PrintWriter` provide the following convenience methods:

`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.
- Class `java.util.Formatter` provides the core support for formatted 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`.

Date/Time (`'t'`, `'T'`):

May be applied to types which are capable of encoding a date or time: `long`, `Long`, `Calendar`, and `Date`. See Time/Date conversions.

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.
`'t','T'`	date/time	Prefix for date and time conversion characters. See Date/Time Conversions.
`'%'`	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 date/time argument types 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.

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

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
```


Format Specifiers for Date/Time Conversions

%[argument_index$][flags][width]conversion

- The optional *argument_index*, *flags* and *width* are defined as for general, character and numeric types.
- The required *conversion* is a *two* character sequence.
  - The first character is `'t'` or `'T'`.
  - The second character indicates the format to be used.

Time Formatting Suffix Characters

- Suffix characters that can be applied to the `'t'` or `'T'` conversions to format time values.

Conversion Specification	Description
`'H'`	Hour of the day for the 24-hour clock, formatted as two digits with a leading zero as necessary i.e. `00 - 23`.
`'I'`	Hour for the 12-hour clock, formatted as two digits with a leading zero as necessary, i.e. `01 - 12`.
`'k'`	Hour of the day for the 24-hour clock, i.e. `0 - 23`.
`'l'`	Hour for the 12-hour clock, i.e. `1 - 12`.
`'M'`	Minute within the hour formatted as two digits with a leading zero as necessary, i.e. `00 - 59`.
`'S'`	Seconds within the minute, formatted as two digits with a leading zero as necessary, i.e. `00 - 60` ("`60`" is a special value required to support leap seconds).
`'L'`	Millisecond within the second formatted as three digits with leading zeros as necessary, i.e. `000 - 999`.
`'N'`	Nanosecond within the second, formatted as nine digits with leading zeros as necessary, i.e. `000000000 - 999999999`.
`'p'`	Locale-specific morning or afternoon marker in lower case, e.g."`am`" or "`pm`".

Conversion Specification	Description
`'z'`	RFC 822 style numeric time zone offset from GMT, e.g. `-0800`.
`'Z'`	A string representing the abbreviation for the time zone. The Formatter's locale will supersede the locale of the argument (if any).
`'s'`	Seconds since the beginning of the epoch starting at 1 January 1970 `00:00:00` UTC, i.e. `Long.MIN_VALUE/1000` to `Long.MAX_VALUE/1000`.
`'Q'`	Milliseconds since the beginning of the epoch starting at 1 January 1970 `00:00:00` UTC, i.e. `Long.MIN_VALUE` to `Long.MAX_VALUE`.

Date Formatting Suffix Characters

- Suffix characters that can be applied to the `'t'` or `'T'` conversions to format dates.

Conversion Specification	Description
`'C'`	Four-digit year divided by `100`, formatted as two digits with leading zero as necessary, i.e. `00 - 99`
`'Y'`	Year, formatted as at least four digits with leading zeros as necessary, e.g. `0092` equals `92` CE for the Gregorian calendar.
`'y'`	Last two digits of the year, formatted with leading zeros as necessary, i.e. `00 - 99`.
`'B'`	Locale-specific full month name, e.g. `"January"`, `"February"`.
`'b'`	Locale-specific abbreviated month name, e.g. `"Jan"`, `"Feb"`.
`'h'`	Same as `'b'`.
`'m'`	Month, formatted as two digits with leading zeros as necessary, i.e. `01 - 13`.
`'A'`	Locale-specific full name of the day of the week, e.g. `"Sunday"`, `"Monday"`
`'a'`	Locale-specific short name of the day of the week, e.g. `"Sun"`, `"Mon"`
`'j'`	Day of year, formatted as three digits with leading zeros as necessary, e.g. `001 - 366` for the Gregorian calendar.
`'d'`	Day of month, formatted as two digits with leading zeros as necessary, i.e. `01 - 31`
`'e'`	Day of month, formatted as two digits, i.e. `1 - 31`.

Common Date/Time Composition Suffix Characters

- Suffix characters that can be applied to the `'t'` or `'T'` conversions for common time/ date compositions.

Conversion Specification	Description
`'R'`	Time formatted for the 24-hour clock as `"%tH:%tM"`
`'T'`	Time formatted for the 24-hour clock as `"%tH:%tM:%tS"`.
`'r'`	Time formatted for the 12-hour clock as `"%tI:%tM:%tS %tP"`. The location of the morning or afternoon marker (`'%tp'`) may be locale-dependent.
`'D'`	Date formatted as `"%tm/%td/%ty"`.
`'F'`	ISO 8601 complete date formatted as `"%tY-%tm-%td"`.
`'c'`	Date and time formatted as `"%ta %tb %td %tT %tZ %tY"`, e.g. `"Sun Jul 20 16:17:00 EDT 1969"`.

Examples: Formatted Time and Date

`Calendar myCalender = Calendar.getInstance();`	
`// Formatting the hour`	
`out.printf("Hour(00-24):%tH%n", myCalender);`	`Hour(00-24):19`
`out.printf("Hour(01-12):%tI%n", myCalender);`	`Hour(01-12):07`
`out.printf("Hour(0-23):%tk%n", myCalender);`	`Hour(0-23):19`
`out.printf("Hour(1-12):%tl%n", myCalender);`	`Hour(1-12):7`
`// Formatting the minutes`	
`out.printf("Minutes(00-59):%tM%n", myCalender);`	`Minutes(00-59):33`
`// Formatting the seconds`	
`out.printf("Seconds(00-60):%tS%n", myCalender);`	`Seconds(00-60):51`
`out.printf("Nanoseconds(000000000-999999999):%tN%n", myCalender);`	`Nanoseconds(000000000-999999999):912000000`
`// PM/pm and AM/am`	
`out.printf("%tp%n", myCalender);`	`pm`
`out.printf("%Tp%n", myCalender);`	`PM`

Examples: Formatted Time and Date (cont.)

`// Formatting the year`	
`out.printf("%tC%n", myCalender);`	`20`
`out.printf("%tY%n", myCalender);`	`2005`
`out.printf("%ty%n", myCalender);`	`05`
`// Formatting the month`	
`out.printf("%tB%n", myCalender);`	`januar`
`out.printf("%tb%n", myCalender);`	`jan`
`out.printf("%th%n", myCalender);`	`jan`
`out.printf("%tm%n", myCalender);`	`01`
`// Formatting the day`	
`out.printf("%tA%n", myCalender);`	`fredag`
`out.printf("%ta%n", myCalender);`	`fr`
`out.printf("%tj%n", myCalender);`	`007`
`out.printf("%td%n", myCalender);`	`07`
`out.printf("%te%n", myCalender);`	`7`
`// Composite Date/Time Format Conversions`	
`out.printf("%tR%n", myCalender);`	`12:11`
`out.printf("%tT%n", myCalender);`	`12:11:56`
`out.printf("%tr%n", myCalender);`	`00:11:56 PM`
`out.printf("%tD%n", myCalender);`	`01/07/05`
`out.printf("%tF%n", myCalender);`	`2005-01-07`
`out.printf("%tc%n", myCalender);`	`fr jan 07 12:11:56 CET 2005`

Examples: Formatted Time and Date (cont.)

```
// Misc. usage.
// Note that the specifiers refer to the same argument in the format string.
out.printf("%1$tm %1$te %1$tY%n", myCalender);
out.printf(US, "The world will end on %1$tA, %1$te. %1$tB %1$tY" +
               " at %1$tH:%1$tM:%1$tS.%n", myCalender);
out.printf("The world will end on %1$tA, %1$te. %1$tB %1$tY at %1$tH:%1$tM:%1$tS.%n",
           myCalender);

Calendar birthdate = new GregorianCalendar(1949, MARCH, 1);
out.printf("Author's Birthday: %1$tD%n", birthdate);
```

Output:

```
01 7 2005
The world will end on Friday, 7. January 2005 at 12:22:25.
The world will end on fredag, 7. januar 2005 at 12:22:25.
Author's Birthday: 03/01/49
```


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
- Delimiters-ignoring Parsing Methods
- 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()`
`boolean hasNext(Pattern pattern)`
`boolean hasNext(String pattern)`
	`String next()`
	`String next(Pattern pattern)`
	`String next(String pattern)`

The first method returns `true` if this scanner has another token in its input.	The first method scans and returns the next complete token from this scanner.
The second method returns `true` if the next token matches the specified pattern.	The second method returns the next string in the input that matches the specified pattern.
The third method returns `true` if the next token matches the pattern constructed from the specified string.	The third method returns the next token if it matches the pattern constructed from the specified string.
`boolean hasNext`*IntegralType*`()` `boolean hasNext`*IntegralType*`(int radix)`	*IntegralType*`' next`*IntegralType*`()` *IntegralType*`' next`*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.	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`.	

Lookahead Methods	Parsing the Next Token
`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`.	
`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.

Delimiters-ignoring Parsing Methods

`Scanner skip(Pattern pattern)`

Skips input that matches the specified pattern, ignoring delimiters.

`Scanner skip(String pattern)`

Skips input that matches a pattern constructed from the specified string.

`String findInLine(Pattern pattern)`

Attempts to find the next occurrence of the specified pattern ignoring delimiters.

`String findInLine(String pattern)`

Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.

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);`	`The`
`  while (lexer.hasNext())`	`world`
`    out.println(lexer.next());`	`will`
`  lexer.close();`	`end`
`}`	`today.`
`{`	
`  String input = "123 45,56 false 567 722 blabla";`	
`    Scanner lexer = new Scanner(input);`	
`    out.println(lexer.hasNextInt());`	`true`
`    out.println(lexer.nextInt());`	`123`
`    out.println(lexer.hasNextDouble());`	`true`
`    out.println(lexer.nextDouble());`	`45.56`
`    out.println(lexer.hasNextBoolean());`	`true`
`    out.println(lexer.nextBoolean());`	`false`
`    out.println(lexer.hasNextInt());`	`true`
`    out.println(lexer.nextInt());`	`567`
`    out.println(lexer.hasNextLong());`	`true`
`    out.println(lexer.nextLong());`	`722`
`    out.println(lexer.hasNext());`	`true`
`    out.println(lexer.next());`	`blabla`
`    out.println(lexer.hasNext());`	`false`
`    lexer.close();`	
`}`	

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";`	`Input: "123 45,56 false 567 722 blabla"`
`   String delimiters = "default";`	
`   parse(input, delimiters, INT, DOUBLE, BOOL, INT, LONG, STR);`	`Delimiters: (default)123`
`}`	**`45.56`**
	`false`
	`567`
	`722`
	`blabla`
`{  `**`// Note the use of backslash to escape characters in regexp.`**	
`   String input = "2004 | 2 | true";`	`Input: "2004 | 2 | true"`
`   String delimiters = "\\s*\\\|\\s*";`	`Delimiters: (\s*\\|\s*)`
`   parse(input, delimiters, INT, INT, BOOL);`	`2004`
`}`	`2`
	`true`
`{  `**`// Another example of a regexp to specify delimiters.`**	
`   String input = "Always = true | 2 $ U";`	`Input: "Always = true | 2 $ U"`
`   String delimiters = "\\s*(\\\||\\$|=)\\s*";`	`Delimiters: (\s*(\\||\$|=)\s*)`
`   parse(input, delimiters, STR, BOOL, INT, STR);`	`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.
}
```


Using Backslash as a Delimiter

- The sequence \\ specifies a backslash in a regexp.
- In a Java string each backslash must be escaped by a backslash.
- Thus four backslashes are required to specify a backslash as a delimiter in a regexp: "\\\\" .

`String input = "C:\\Program Files\\3MM\\MSN2Lite" +`
`               "\\Help";`
 	
`String delimiters = "\\\\";`
`out.println("Input: \"" + input + "\"");`
`out.println("Delimiters: (" + delimiters + ")");`
`Scanner lexer = new Scanner(input)`
`               .useDelimiter(delimiters);`
`while (lexer.hasNext())`
`  out.println(lexer.next());`
`lexer.close();`
	
	`Input: "C:\Program Files\3MM\MSN2Lite\Help"`
	`Delimiters: (\\)`
	
	`C:`
	`Program Files`
	`3MM`
	`MSN2Lite`
	`Help`

Using Delimiters and Patterns with a Scanner

- In the example below, the delimiter is a backslash as shown at (1).
- The pattern specifies tokens that are a sequence of lower case and upper case letters, as shown at (2) .
  - `[a-z[A-Z]]` specifies *union of classes*, in this case, lower case (`a-z`) and upper case (`A-Z`) letters.
  - The *positive Kleene closure* (+) specifies one or more occurrence of its operand, in this case, lower case and upper case letters.

```
String input = "C:\\Program Files\\3MM\\MSN2Lite" +
               "\\Help";
String delimiters = "\\\\";                        // (1) Delimiter is \.
String patternStr = "[a-z[A-Z]]+";                // (2) Pattern for tokens to match.
out.println("Input: \"" + input + "\"");
out.println("Delimiters: (" + delimiters + ")");
out.println("Pattern: (" + patternStr + ")");
```


- The loop at (3) ensures that all tokens are handled, regardless whether they match the pattern or not.
  - A token that does not match the pattern is explicitly skipped at (5).

```
Scanner lexer = new Scanner(input).useDelimiter(delimiters);
while (lexer.hasNext())                           // (3) End of input?
  if (lexer.hasNext(patternStr))                  // (4) Token matches pattern?
      out.println(lexer.next(patternStr));        // (4) Parse the token
  else
      lexer.next();                               // (5) Skip unmatched token.
lexer.close();
```

Output:

```
Input: "C:\Program Files\3MM\MSN2Lite\Help"
Delimiters: (\\)
Pattern: ([a-z[A-Z]]+)
Help
```

A Simple Word Frequency Lexer

```
Scanner lexer = new Scanner(new File("FormattedInput.java"));
String wordPattern = "[a-z[A-Z]]+";
Map<String, Integer> freqM = new TreeMap<String, Integer>();
while (lexer.hasNext())
  if (lexer.hasNext(wordPattern)) {
    String word = lexer.next(wordPattern);
    Integer freq = freqM.get(word);
    freqM.put(word, freq == null ? 1 : freq + 1);
  }
  else
    lexer.next();
lexer.close();
out.println("Word count: " + freqM.size());
out.println(freqM);
```

Output:

```
Word count: 56
{A=1, Backslash=1, Delimiter=2, Delimiters=1, End=1, Formatted=1, FormattedInput
=1, Frequency=1, IOException=1, Input=1, Integer=1, Lexer=1, Logical=1, Parse=1,
 Pattern=1, Patterns=2, Scanner=6, Simple=1, Skip=1, String=9, Token=1, Using=3,
 Word=1, a=2, and=1, as=1, class=1, delimiters=4, else=2, error=1, for=1, ...}
```