Introduction to Programming (Java)

Roman Szturc (roman.szturc@vsb.cz)
Zdenek Sawa (zdenek.sawa@vsb.cz)

Department of Computer Science
VSB – Technical University of Ostrava

Info for Students

Lectures: Zdenek Sawa (e-mail: zdenek.sawa@vsb.cz, room: A1006)

Consultations: Wednesday 9:00 – 10:30

Exercises: 40 points

Num.	Date	Points
1.	08.–12.03.	5
2.	22.–26.03.	6
3.	05.–09.04.	6
4.	19.–23.04.	7
5.	03.–07.05.	8
6.	17.–21.05.	8

Exam: 60 points

WWW: http://www.cs.vsb.cz/java/

See also: http://www.cs.vsb.cz/benes/vyuka/upr/index.html

References

- Sun Microsystem, Inc., **The Source for Java Technology**,
  http://java.sun.com

- J. Gosling, B. Joy, G. Steele, G. Bracha: **The Java Language Specification**
  http://java.sun.com/docs/books/jls/index.html

- JavaTM 2 Platform, Standard Edition, v 1.4.2 API Specification,
  http://java.sun.com/j2se/1.4.2/docs/api/index.html

- B. Eckel, **Thinking in Java**,
  http://www.mindview.net/Books/TIJ

- JavaWorld.com, an IDG Communications company, **JavaWorld**,
  http://www.javaworld.com

Java – Overview

Java is a general-purpose language with the following features:

- is object-oriented (class-based)
- is cross-platform
- is strongly typed
- has garbage collection
- supports concurrecy
- supports exceptions
- security is considered

It was created by James Gosling from Sun Microsystems in 1990. Original name was Oak and it was intended for use in embedded consumer-electronic applications.

Later in 1993 it was renamed to Java and retargeted to Internet applications.

First official implementation (JDK 1.0) was released in 1996.

Creating Java Program (Step 1)

Source code creation

In the Java, each method (function) and variable exists within a class or an object. The Java does not support global functions or variables. Thus, the skeleton of any Java program is a class definition.

Every Java application must contain a `main()` method. The method is invoked when the application is executed by a Java interpreter.

```
public class HelloWorld {
   public static void main(String[] arguments) {
      System.out.println("Hello, world...");
   }
}
```

Note: A Java class source code **must** be stored in file which name starts with the **class name** appended with the **.java**, so the previous example must be stored in a file called `HelloWorld.java`.

Creating Java Program (Step 2)

Compilation

Compilation transfers Java source code into Java bytecode. There is a lot of compilers available. The most often used ones are `javac` and `jikes`.

File name containing the source code is passed to the compiler. Result bytecode is stored into file whose name is the **class name** appended by the **.class** suffix.

```
$ javac HelloWorld.java
```

Note: Each compiler provides its specific options. The most important are `-classpath` <path> and `-g`.

Creating Java Program (Step 3)

Running

Java bytecode can be executed using a Java interpreter. The **class name** is passed as an argument to the interpreter.

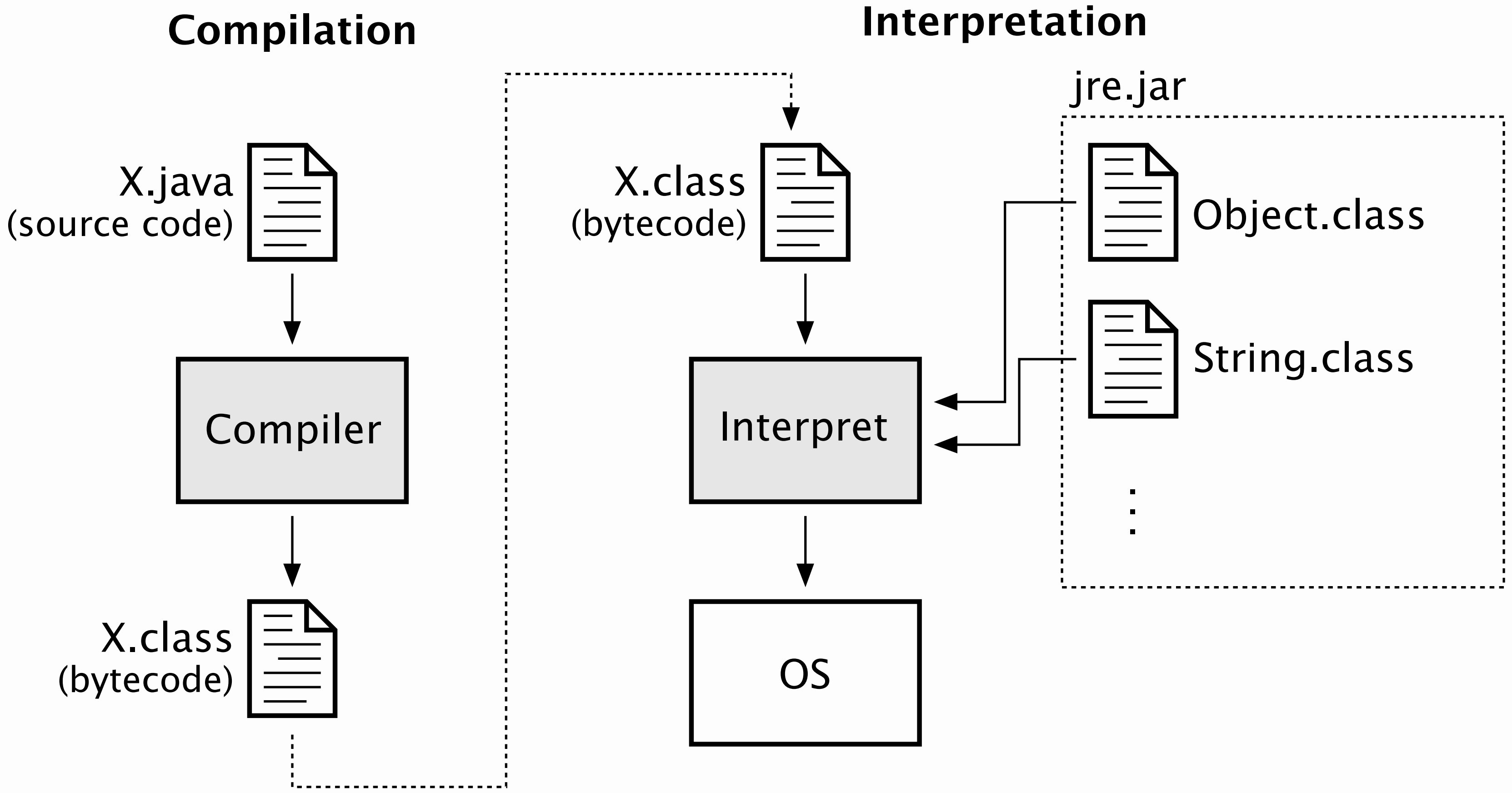
Environment variable CLASSPATH plays important role while executing the Java bytecode. It holds list of directories and libraries containg bytecodes being executed.

```
$ java HelloWorld
Hello, world...
```

Note: It is convenient to include the "." (dot) in the CLASSPATH. The "." ensures that bytecodes in the current working directory are successfully found by interpreter.

Java Environment

Java combines compilation and interpretation techniques. Java source code is first compiled into an intermediate language called *bytecode*. The bytecode helps make “write once, run anywhere” possible.



Note: Compilation happens just **once**; interpretation occurs **each time** the program is executed.

Bytecode

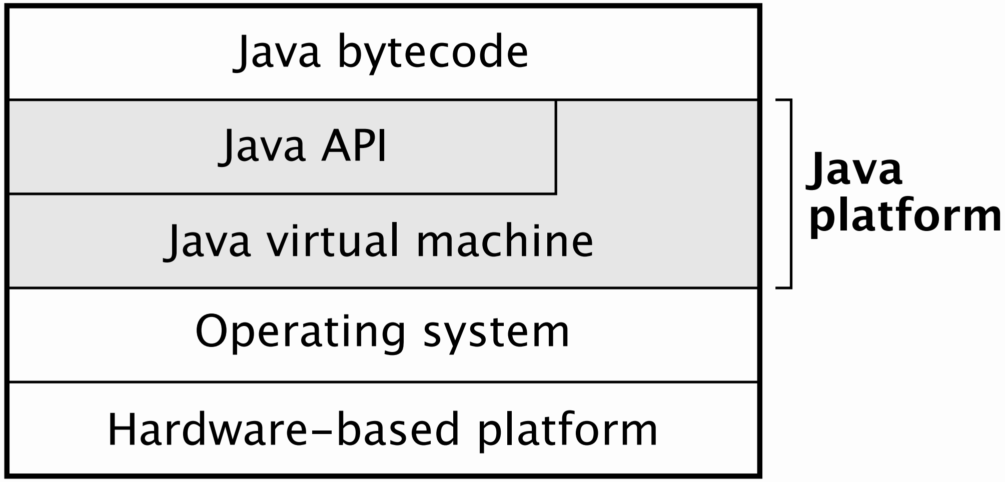
An example of bytecode produced by javap utility.

```
$ javap -c HelloWorld

Compiled from HelloWorld.java
public class HelloWorld extends java.lang.Object {
    public HelloWorld();
    public static void main(java.lang.String[]);
}

Method HelloWorld()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 getstatic #2 <Field java.io.PrintStream out>
   3 ldc #3 <String "Hello, world...">
   5 invokevirtual #4 <Method void println(java.lang.String)>
   8 return
```

The Java Platform

A **platform** is the hardware or software environment in which a program runs. The **Java platform** differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms. The Java platform has two components:

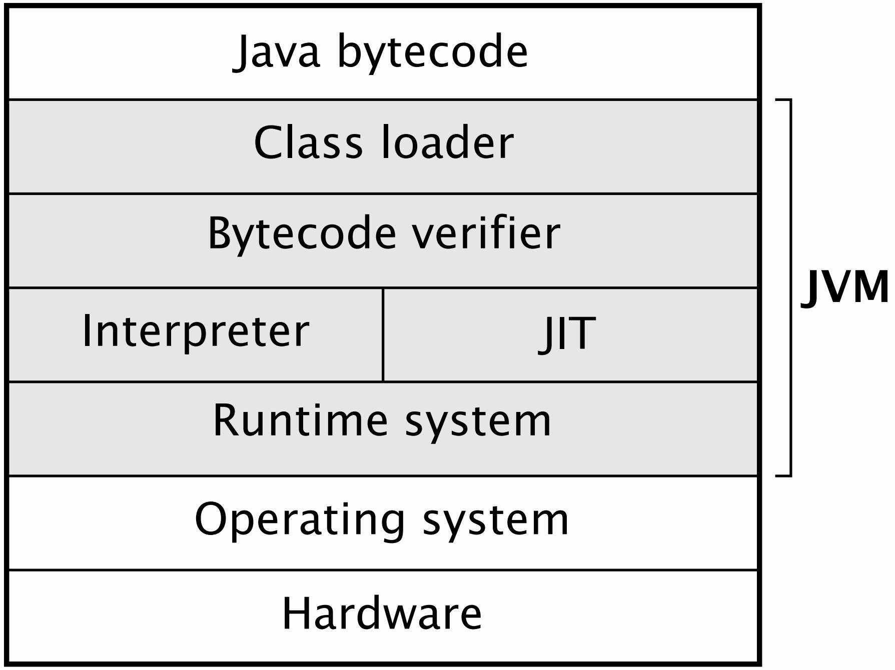
- Java virtual machine,
- Java application programming interface (API).



Note: The Java API is a large collection of **ready-made** software components that provide many useful capabilities. The Java API is grouped into libraries of related **classes** and **interfaces**; these libraries are known as **packages**.

Java Virtual Machine

Java bytecode is machine code for the **Java Virtual Machine** (JVM). Every Java interpreter is an implementation of a JVM.



Java bytecode

Class loader

Bytecode verifier

Interpreter | JIT

Runtime system

Operating system

Hardware

JVM

Note: A Java program can be compiled into bytecodes on any platform that has a Java compiler. The bytecodes can then be run on any implementation of the JVM, regardless of operating system or hardware platform.

Syntax and Semantics

Syntax – describes what constructions are possible in the language, what is a correct program and what is not

Semantics – describes what these constructions mean, e.g., what computer does when it performs given commands

Java distinguishes between two types of errors:

Compile time errors: They are produced by compiler. These errors are either

- **syntax** – for example missing **;**
- **semantic** – for example assignment between incompatible types

Run time errors: They are produced by Java Virtual Machine during execution of a program. Java contains no dangerous constructions, always an **exception** is generated.

Overview of the Syntax of Java

- The programs consist of **classes**.
- Each class consists of definitions of data and instructions for a certain kind of **objects**.
  - **fields (attributes):** data of an object
  - **methods:** instructions that manipulate with these data (also called **functions** or **procedures** in other languages)
- A method consists of a **header** that defines its name, arguments, return type, ... and **body** that contains **statements**.
- Statements manipulate with **data** stored in **variables** – either in **fields** or in **local variables**.
- Execution of statements includes evaluation of **expressions**. The values of expressions are then assigned into variables.
- Each variable, value and expression is of some **type**.
- On the lowest level a program is a sequence of **lexical elements (tokens)**.

Lexical Elements

White space characters and comments are ignored:

- **white space characters**: space (SP), horizontal tab (HT), form feed (FF), newline (LF), carriage return (CR)
- **comments**: `/* this is a comment */`

Basic types of lexical elements (tokens) are:

- **identifiers**: `x` `dist1` `System9` `number_of_elements`
- **keywords**: **`while`** **`float`** **`int`** **`public`** **`class`**
- **literals**: `124` **`true`** `'d'` `"hello"`
- **separators**: `(` `)` `{` `}` `[` `]` `;` `:` `,` `.`
- **operators**: `+` `-` `*` `/` `&&` `=` `*=` `<` `>>=`

Literals

- **integer literals:**
  ```
  0   237L   033   0xDadaCafe   1996  0x00FF00FF
  ```
- **floating point literals:**
  ```
  1e1  2.  .3   0.0   3.14f  1.213e-9   1E137D
  ```
- **boolean literals:**
  ```
  true   false
  ```
- **character literals:**
  ```
  'a'  '%'  '\t'  '\\'  '\''  '\177'  '\u03a9'
  ```
- **string literals:**
  ```
  ""  "\""  "This is a string."   "\r\n"
  ```
- **the null literal:**
  ```
  null
  ```

Possible **escape sequences** in character and string literals:

- `\b  \t  \n  \f  \r  \"  \'  \\  \177  \u2B97`

Keywords

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

Comments

Java supports three kinds of comments:

- **One-line comment** – the compiler ignores everything from the “//” to the end of line.

```
// This is a one-line comment.
```

- **Multi-line comment** – the compiler ignores everything from the “/*” to an occurrence of “*/”.

  Note: “/*/” is not a valid comment.

```
/* This is a comment that
   continues across lines. */
```

- **Documentation comment** – the compiler ignores everything from the “/**” to an occurrence of “*/”. `javadoc` tool generates documentation based on content of the comment.

```
/** This is a documentation comment.
 *  The comment may contain html tags as well as special
 *  tags that begin with the '@' sign. */
```

Types, Values and Variables

Variables are used by program to hold data. Each variable used in program must be explicitly specified by its **data type** and **name**. Java has two kinds of data types: **reference** and **primitive**.

- **Primitive**

  A variable of primitive type contains a **single value** of the appropriate size and format for its type: a number, a character or a boolean value.

  ```
  boolean b = true;
  int i = 456;
  float f = 2.71828;
  ```

- **Reference**

  The value of a reference type variable, in contrast to that of a primitive type, is a **reference** to (an address of) an object or an array.

  ```
  Hashtable h = new Hashtable();
  int[] a = new int[20];
  ```

Integral Types and Values

Type	Range	Size [bits]
byte	-128..127	8
short	-32768..32767	16
int	-2147483648..2147483647	32
long	-9223372036854775808..9223372036854775807	64
char	0..65535	16

Possible operations on integer values are:

- the comparison operators (<, <=, >, >=, ==, !=)
- the unary plus and minus (+, -)
- the binary arithmetic operators (+, -, *, /, %)
- the prefix and postfix increment and decrement operators (++, --)
- the signed and unsigned shift operators (<<, >>, >>>)
- the bitwise complement operator (~)
- the integer bitwise operators (&, |, ^)

Floating-Point Types and Values

The floating-point values are numbers of the form $sm2^e$ where

Type	s	m	e	Size [bits]
`float`	-1, 1	0..2^{24}-1	-149..104	32
`double`	-1, 1	0..2^{53}-1	-1075..970	64

Type	Min. value	Max. value
`float`	1.40239846e-45f	3.40282347e+38f
`double`	4.94065645841246544e-324	1.79769313486231570e+308

Possible operations on floating-point values are:

- the comparison operators (<, <=, >, >=, ==, !=)
- the unary plus and minus (+, -)
- the binary arithmetic operators (+, -, *, /, %)
- the prefix and postfix increment and decrement operators (++, --)

The Boolean Type and Values

The type **boolean** has two possible values: **true** and **false**

Possible operations on floating-point values are:

- the relational operators (==, !=)
- the logical complement operator (!)
- the binary logical operators (&, |, ^)
- the conditional-and and conditional-or operators (&&, ||)
- the ternary conditional operator (?:)

Boolean expressions determine the control flow in several kinds of statements:

- the **if** statement
- the **while** statement
- the **do** statement
- the **for** statement

The Arithmetic Operators

The arithmetic operators refer to the standard mathematical operators: addition, subtraction, multiplication, division and modulus.

Op.	Use	Description
+	`x + y`	adds x and y
-	`x - y`	subtracts y from x
*	`x * y`	multiplies x by y
/	`x / y`	divides x by y
%	`x % y`	computes the reminder of dividing x by y

Examples:

```
i + 1        (x * y) % 5        b * b - 4 * a * c
```

Some remarks for integer arithmetic operators:

- The result contains only the low-order bits of the mathematical result in case of the arithmetic overflow.

. Unary Operators

Java's unary operators can use either **prefix** or **postfix** notation.

Operator	Use	Description
+	+op	promotes op to `int` if it is a `byte`, `short` or `char`
-	-op	arithmetically negates op
++	++op	increments op by 1; evaluates to value of op before the incrementation
++	op++	increments op by 1; evaluates to value of op after the incrementation
--	--op	decrements op by 1; evaluates to value of op before the decrementation
--	op--	decrements op by 1; evaluates to value of op after the decrementation

Examples:

```
-x        +(x * y)        i++        a[j--]++
```

Examples of use of ++ and --

Code:

```
int x = 5; int y;
y = x++;
```

Results:

```
x = 6     y = 5
```

Code:

```
int x = 5; int y = 11; int z;
z = --x;
x = 2 * (y++ + 3) - x;
```

Results:

```
x = 24    y = 12     z = 4
```

Relational Operators

Relational operators generate a **boolean** result.

Operator	Use	Returns true if
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Examples:

```
i + 1 < n          x == h[2*i+1]          a != b
```

Conditional Operators

Relational operators are often used with conditional operators.

<table>
<tr><th>Operator</th><th>Use</th><th>Returns true if</th></tr>
<tr><td>&&</td><td>op1 && op2</td><td>op1 and op2 are both true, conditionally evaluates op2</td></tr>
<tr><td>||</td><td>op1 || op2</td><td>either op1 or op2 is both true, conditionally evaluates op2</td></tr>
<tr><td>!</td><td>!op1</td><td>op1 is false</td></tr>
</table>

Examples:

```
!(n >= 0)
(i < n) && (a[i++] > 0)
```

If (i>=n) then the value of i is not changed. If (i<n) then i is incremented by 1.

Bitwise Operators

The bitwise operators allow to manipulate individual bits in an integral primitive data type. Bitwise operators perform boolean algebra on the corresponding bits in the two arguments to produce the result.

Operator	Use	Operation
&	op1 & op2	bitwise and
|	op1 | op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~op	bitwise complement

Examples:

```
0x36 & 0x0F     0x06     (00110110 & 00001111)
0x36 | 0x80     0xB6     (00110110 | 10000000)
0x36 ^ 0x07     0x31     (00110110 ^ 00000111)
```

Shift Operators

Shift operator shifts the bits of the left-hand operand over by the number of positions indicated by the right-hand operand. The shift occurs in the direction indicated by the operator itself.

Operator	Use	Operation
>>	op1 >> op2	shift bits of op1 right by distance op2
<<	op1 << op2	shift bits of op1 left by distance op2
>>>	op1 >>> op2	shift bits of op1 right by distance op2

```
0x36 << 2      0xD8      (00110110  ->  11011000)

-1             -1 (decimal)
-1 >> 1        -1 (decimal)
-1 >>> 1       2147483647 (decimal)

-1             11111111111111111111111111111111 (binary)
-1 >> 1        11111111111111111111111111111111 (binary)
-1 >>> 1       01111111111111111111111111111111 (binary)
```

Ternary Operator (?:)

The ternary operator allows to avaluate expresseion in two diferrent ways depending on some condition.

The expression is of the form:

```
cond ? expr1 : expr2
```

The boolean condition cond is evaluated first. If it is **`true`** then expr1 is evaluated and the resulting value is the value of the whole expression. When cond evaluates to **`false`** then expr2 is evaluated and the resulting value is the value of the whole expression.

Example:

```
(n > 1) ? (a + b) : (a * b)
```

When (n>1) then the result is (a+b), otherwise the result is (a*b).

Assignment Operators

The basic form of assignment is

```
expr1 = expr2
```

Evaluation:

1. The left hand side (expr1) is evaluated. It must by an **lvalue** – a variable, an element of an array, a field.
2. The right hand side (expr2) is evaluated.
3. The value of the right hand side is stored into the place denoted by the left hand side.
4. The value of the whole expression is the value of the right hand side.

Examples of assignment expressions:

```
x = (z + y) * a[i]
a[i++] = x + y
```

Assignment Operators (cont.)

Examples of assignment statements:

```
x = (z + y) * a[i];
a[i++] = x + y;
```

Note that an assignment expression is not the same thing as an assignment statement.

The following construction is legal, but the resulting code is not very clear:

```
int y, x;
x = 3 * (y = 2) + 1;
```

The results are:

```
 x = 7        y = 2
```

Compound Assignment Operators

There other assignment operators of the form op= where op is some binary operator:

```
*=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
```

The meaning of

```
expr1 op= expr2
```

is the same as

```
expr1 = expr1 op expr2
```

except that expr1 is evaluated only once.

For example, the statement `x *= 6;`
has the same effect as `x = x * 6;`

Notice that `a[i++] += 3;`
is not the same as `a[i++] = a[i++] + 3;`

Cast Expression

The following assinment between variables of different types is possible:

```
byte b; int i;
    .
    .
    .
i = b;
```

The following assignment is **illegal**:

```
b = i;
```

It can be assigned using the cast of the form

```
(type)expr1
```

which transforms the value of expr1 to the type type as in the following code:

```
b = (byte)i;
```

Priority of Operators

Operators ordered by priority (from lowest to highest):

Pr.	Operators
1.	()
2.	[], postfix ++ and --
3.	unary +, unary -, ~, !, cast, prefix ++ and --
4.	*, /, %
5.	+, -
6.	<<, >>, >>>
7.	<, >, <=, >=, instanceof
8.	==, !=
9.	&
10.	^
11.	|
12.	&&
13.	||
14.	?:
15.	=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=

Associativity of Operators

Most binary operators are associative to the left.

For example

```
a + b + c
```

has the same meaning as

```
(a + b) + c
```

An exception are the asignment operators that are associative to the right.

For example

```
a = b = c
```

has the same meaning as

```
a = (b = c)
```

Statements

One of the basic types of statements is an **assignment statement**:

```
a = b + c;
```

Assignment statement must end with semicolon (;).

Some other types of expressions can be also used as statements:

```
i++;
sum(a, b);
```

A **declation** can be also used as a statement:

```
int i;
double x, y, z;
```

A declaration can be combined with an assignment of an initial value:

```
int i = 4;
double x = 46.3, y, z = i * 2.0;
```

Blocks

Blocks are sequences of statements enclosed between { and }.

Example:

```
{
    a = 3;
    int b = a + 1;
    a = b * 2;
}
```

The **scope** of a declation of a local variable is from the place where it is declared to the end of the enclosing block.

A block can be used in any place where a single statement can be used.

Branching Statement

The **if-else** statement is probably the most basic way to control program flow.

```
if (value > value2) {
   result = 1;
}
else if (value1 < value2) {
   result = -1;
}
else {
   result = 0;
}
```

Similarly we can use:

```
if (value > value2) result = 1;
else if (value < value2) result = -1;
else result = 0;
```

Iteration Statements

Java provides three iteration statements. The statements repeat their bodies until controlling expression evaluates to **false**.

- **while**

```
int i = 0;
while (++i < 2)
   System.out.println("i: " + i);
```

- **do-while**

```
int i = 0;
do {
   System.out.println("i: " + i);
} while (++i < 2)
```

- **for**

```
int powerOfTwo = 1;
for (int i = 0; i < 16; i++)
   powerOfTwo <<= 1;
```

Driving Iteration Statements

Inside the body of any of the iteration statements flow of the loop can be controlled using **break** and **continue** statements. **break** quits the loop without executing the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration.

```
int i = 0;
while (true) {
   if (i > 20)
      break;
   if (i++ % 7 == 0)
      continue;
   i += 2;
}
```

Driving Iteration Statements

The **break** and **continue** normally only alter the closest looping structures. If there are nested statements, **labeled break** and **continue** can be used to alter outer looping structures.

```
int i = 0;
outer:
while (true) {
   while (true) {
      i++;
      if (i == 1)
         break;
      if (i == 4)
         break outer;
   }
   while (true) {
      i++;
      if (i == 2)
         continue;
      if (i == 3)
         continue outer;
   }
}
```

The switch Statement

The **switch** statement is used to test an **integral expression** against one or more possible cases.

```
char ch;
boolean whitespace;

switch (ch) {
   case ' ':
   case '\n':
   case '\t':
   case '\r':
      whitespace = true;
      break;
   default:
      whitespace = false;
}
```

Array

An **array** is a structure that holds multiple values of the same type. The length of an array is established when the array is created. After creation, an array is a **fixed-length** structure. Array identifier is actually a reference to a true object that holds the references to the other objects.

```
double gears[] = new double[5];

gears[0] = 4.624;
gears[1] = 3.231;
gears[2] = 2.893;
gears[3] = 1.052;
gears[4] = 0.962;
```

Sometimes it is convenient to initialize an array immediately during its declaration.

```
double gears[] = {4.624, 3.231, 2.893, 1.052, 0.962};
```

Note: Although the **new** operator is not presented, the array is allocated dynamically (compiler does it for us).

Array (cont.)

The index of array elements start from 0.

When an array contains n elements, the elements have indexes from 0 to (n-1).

Special attribute `length` contains the number of elements in the array.

When for example

```
int[] a = new int[10];
int n = a.length;
```

The following form of a declaration of an array are possible, they are both equivalent:

```
int[] a;
int a[];
```

Array (cont.)

When we declare a variable such as

```
int[] a = new int[10];
```

it is a **reference** to an array. So after assignment

```
int[] b = a;
```

both a and b point to the same array and when we change a value of a[0], the value of b[0] is also changed.

Multidimentional Array

Multidimentional array is in fact **one-dimentional** array containing arrays as its elements.

```java
final byte EMPTY = 0;
final byte CIRCLE = 1;
final byte CROSS = 2;
byte board[][] = {
   {EMPTY, CIRCLE, CROSS},
   {CIRCLE, EMPTY, CROSS},
   {EMPTY, CIRCLE, EMPTY}
};
for (int i = 0; i < board.length; i++)
   for (int j = 0; j < board[i].length; j++)
      System.out.println("board[" + i + "][" + j + "] = " +
         board[i][j]);
```

Note: The `length` is not a method. The `length` is a property provided by the Java platform for all arrays.

Manipulating Arrays

The `java.lang.System.arraycopy()` method provides efficient copy of data from one array into another.

```
char from[] = {'a', 'b', 'c', 'd', 'e', 'f'};
char to[] = new char[3];
System.arraycopy(from, 2, to, 0, to.length);
```

Note: Destination array must be allocated before `arraycopy()` is called and must be large enough to contain the data being copied.

Method Definition

A definition of a **method** begins with a **header** that if followed by a **body** of the method.

The header has the following format:

```
type name ( args )
```

where

- name is the name of the method
- type is a return type of the method
- args is a comma separated list of arguments, it may be empty

Each argument is of the form type name.

The **body** of a method is a block (i.e., statements enclosed between { and }).

Method Definition (cont.)

The return type may be **void** if the method does not return a value. When the return type is not **void** the method must return a value using the command

```
return expr;
```

An example:

```
int gcd(int a, int b)
{
    while (b != 0) {
        int c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```

If the return type is **void** the following form of the return statement can be used:

```
return;
```

Method Invocation

A method is called using its name that is followed by a comma separated list of parameters between paranthesis.

Any expression can be used as a parameter.

All parameters are evaluated and assigned to the arguments of the method.

A method invocation can be used in expression. The value of the expression corresponding to the method invocation is the return value returned by the method.

An example:

```
int x, y;
x = gcd(24, 18);
y = gcd(x + 1, 36);
```

Object-Oriented Modeling

Object-oriented modeling is a method that models the characteristics of real or abstract objects from application domain using classes and objects.

- **Objects**

  Software objects are modeled after real-world objects in that they too have **state** and **behavior**.

  - A software object maintains its state in one or more **variables** (**attributes**).
  - A software object implements its behavior with **methods** that manipulate these variables.

- **Messages**

  Software objects interact and communicate with each other by sending messages. When object A wants object B to perform one of B's methods, object A sends a message to object B.

Messages

Sometimes, the receiving object needs more information to know exactly what to do. This information is passed along with the message as **parameters**.

Message sending requires the following information:

- the object to which the message is addressed,
- the name of the method to perform,
- any parameters needed by the method.

The sending of a message can have any of the following effects:

- The state of the receiving object is changed.
- Some other actions are performed (including sending another messages to some objects).
- Some information is returned to the sending object.

Examples of Objects

Objects in a program correspond to objects from the application domain.

Information system of a bank: accounts, transactions, clients, other banks

Chess playing program: chess pieces, a chessboard, positions, moves, games, strategies

Action game: monsters, weapons, walls, doors, flying bullets, a score counter

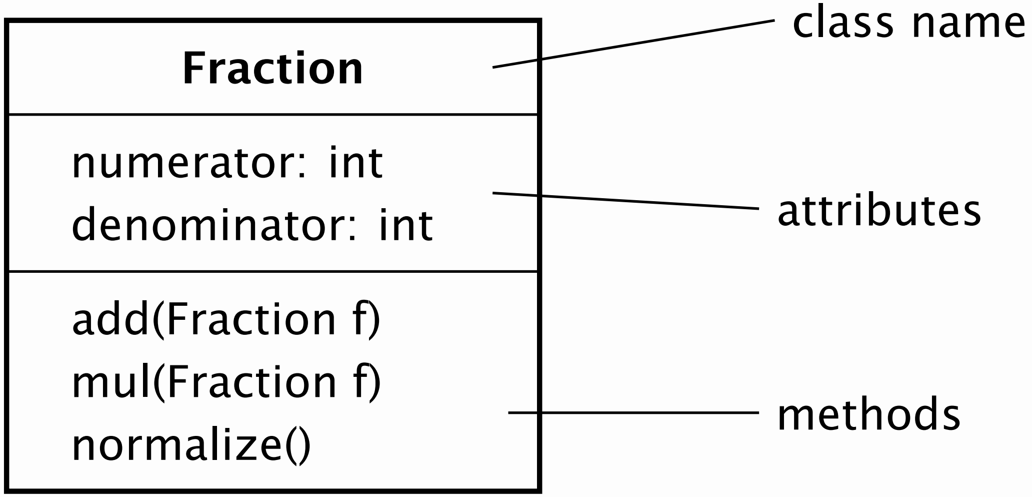
Drawing application: lines, rectangles, circles, arrows, text fields, line styles, line colors

GUI toolkit: windows, buttons, menus, menu items, icons

Class

In the real world, many objects of the same kind exist. Using object-oriented terminology, the objects are **instances** of a **class**. A class is prototype that defines variables and methods common to all objects of a certain kind.

Graphical representation of a class:



Fraction

numerator: int
denominator: int

add(Fraction f)
mul(Fraction f)
normalize()

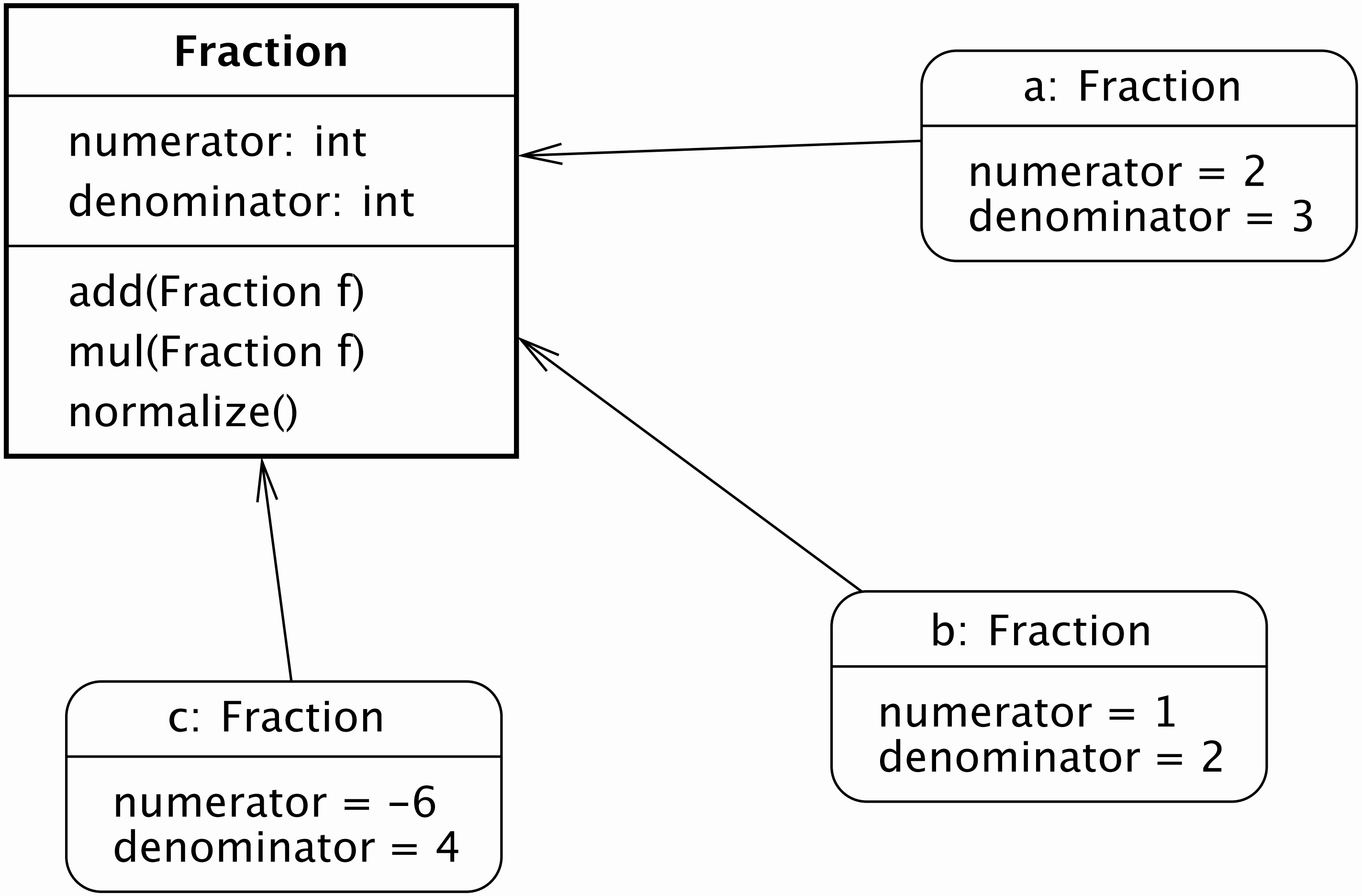
class name

attributes

methods

Instances

Objects are **instances** of the given **class.** Each object has its own **identity.**



Java Class

Java defines classes using the **class** keyword. Definition of a class may contain declarations of variables, definitions of methods or even nested classes.

The order of class members is not important.

```
class Fraction
{
  int  numerator;
  int  denominator;

  void mul(Fraction f)
  {
    numerator   *= f.numerator;
    denominator *= f.denominator;
  }
}
```

Note: Standard convention is that class names start with an upper-case letter and class member names (attributes and methods) start with a lower-case letter.

Java Objects

An instance of a class can be created using **new** operator. It allocates required space on the heap, provides initialization and returns reference to the newly created instance.

```
Fraction a = new Fraction();
```

The attributes of a object can be accessed using the reference to the object and the dot (.).

```
a.numerator = 2;
a.denominator = 3;
```

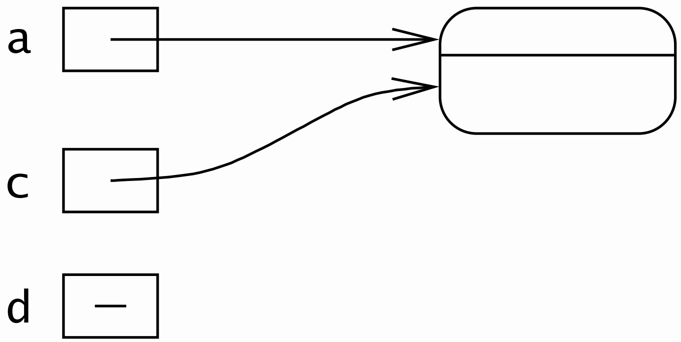
The methods of an object are accessed similarly.

```
Fraction b = new Fraction();
b.numerator = -6;
b.denominator = 4;

a.mul(b);
```

References

The references are just pointers to objects in memory. So after the following assignment the both a and c point to the same object.

```
Fraction c = a;
```



There is a special reference value **null** that denotes reference that does not point to any object.

```
Fraction d = null;
```

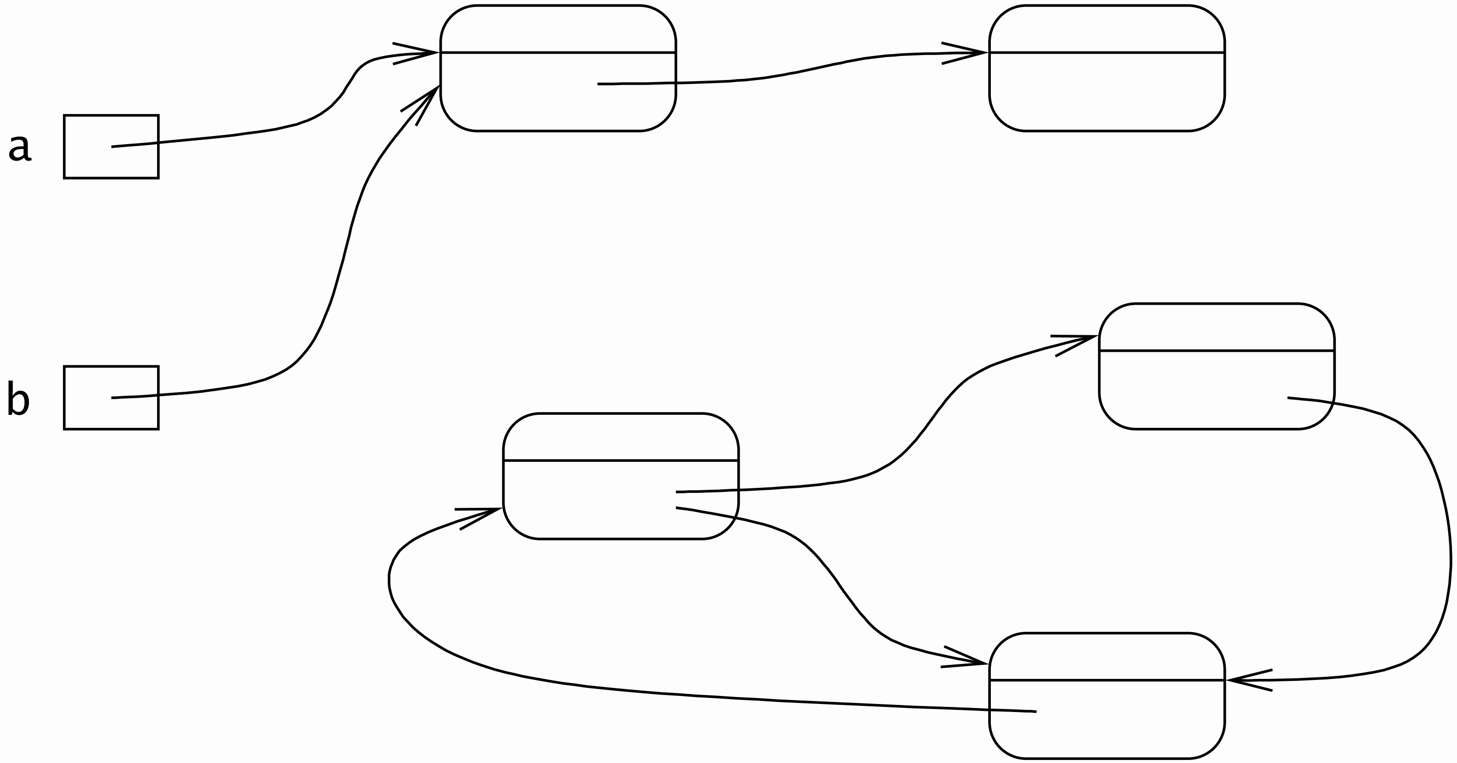
It is an error to access attributes or methods using a reference with the **null** value.

How To Destroy Objects?

There is no need to destroy objects explicitly in Java, since it uses automatic memory management – **garbage collector**.

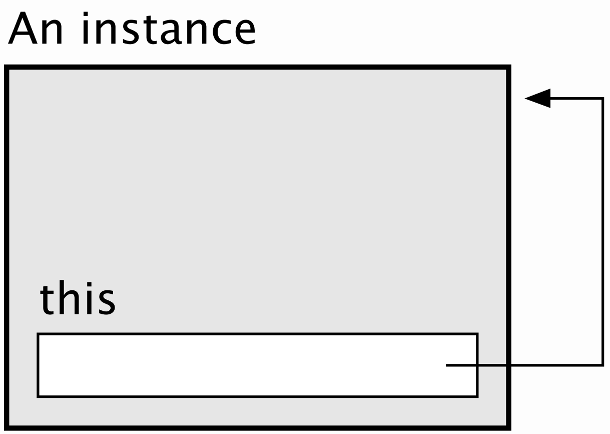
Whenever there is no reference to an object, the object can be destroyed and the memory used by this object is freed.

The garbage collector takes into account circular dependencies.



The this Keyword

The **this** keyword produces the reference to the object the method has been called for.



Examples (it is possible to omit **this** in most cases):

```
Fraction a, b;
  ...
this.numerator = 3;              numerator = 3;
this.denominator = 2;            denominator = 2;
this.add(a);                     add(a);
b.mul(this);
```

Overloading of Methods

The methods in Java can be **overloaded**. This means that there can be more methods with the same name in a class. The methods must differ in the number and types of their parameters.

The method that is actually called is chosen depending on the number and types of parameters.

The return types of overloaded methods need not be all the same.

```
class Fraction {
      ...

    void add(int x) { ... }

    void add(int num, int den) { ... }

    void add(Fraction f) { ... }

      ...
}
```

Encapsulation

Some attributes and methods can be marked as **private**. Such attributes and methods can be accessed only from methods in the class where they are defined. An attempt to access them from methods in other classes produces a compile-time error.

Attributes and methods can be also marked as **public**. Such attributes and methods can be accessed from any other class.

```
class Fraction
{
    private int numerator;
    private int denominator;

    public void set(int num, int den) { ... }

    public int getNumerator() { ... }

    public int getDenominator() { ... }

      ...
}
```

Encapsulation (cont.)

Encapsulation (also **information hiding**) is the separation of the external aspects of an object (accessible from other objects) from the internal implementation details (which are hidden from other objects).

The implementation of an object can be changed without affecting the other parts of an application that use it.

In a purely object-oriented design the attributes of an object are always private and the only way to access them is through the methods that manipulate them.

The use of keywords **`private`** and **`public`** allows to promote encapsulation.

> **Note:** If none of keywords **`private`** and **`public`** is used then the member can be accessed from other classes. However, there are differences between using and not using the keyword **`public`**. They will be discussed later. There is also a keyword **`protected`** that will be discussed later too.

Initialization of an Object

After the creation of a new object (using **new**), all its attributes are set to zero:

- numeric values (**int**, **long**, **char**, **float**, ...) are set to 0
- boolean values are set to **false**
- references (to objects and to arrays) are set to **null**

It is possible to set attributes to some specified values using explicit initialization:

```
class Fraction
{
    int numerator = 0;
    int denominator = 1;

      ...
}
```

Constructors

A more elaborate initialization of an object can be implemented using **constructors**.

Constructor resemble methods, but there are some differences:

- The name of a class must be used as a name of a constructor.
- Constructors can not return values.
- A constructor can be invoked only in the time of the creation of an object (using **new**).

```
class Fraction {
      ...
    Fraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

```
Fraction a = new Fraction(3, 5);
a.add(new Fraction(1, 3));
```

Constructors (cont.)

Some additional remarks concerning constructors follow:

- Constructors may be overloaded, similarly as methods.
- Constructors can be marked as **public**, **protected** and **private**, similarly as methods.
- When no constructor is defined then the default constructor that does nothing is defined automatically, as if the following empty constructor would be put in the code:

```
class Fraction
{
    Fraction() { }
      ...
}
```

- If there is explicitly defined at least one constructor, the default empty constructor is not defined automatically.

Constructors (cont.)

Constructors can call other constructors. The keyword **this** can be used for this purpose. An invocation of another constructor can be used only as the first statement of a calling constructor's body.

```
class Fraction {

    Fraction() {
        this(0);
    }

    Fraction(int x) {
        this(x, 1);
    }

    Fraction(int num, int den) {
        numerator = num;
        denominator = den;
    }

      ...
}
```

Static Members

Variables and methods defined by a class can be of two types: **instance** and **class** (**static**). Class members are distinguished from instance ones by the **`static`** keyword.

- **Instance Members**
    - **Instance Variable**
      Any item of data that is associated with a particular object. Each instance of a class has its **own copy** of the instance variables defined in the class.
    - **Instance Method**
      Any method that is invoked with respect to an instance of a class.
- **Class Members**
    - **Class Variable**
      A data item associated with a particular class as a whole, not with particular instances of the class.
    - **Class Method**
      A method that is invoked without reference to a particular object. Class methods affect the class as a whole, not a particular instance of the class.

Static Members (cont.)

- **Instance Variables and Methods**
  They can be accessed only using a reference to some object.

```
obj.variable,  obj.method()
```

- **Class (Static) Variables and Methods**
  The keyword **static** is used to denote them. There is always exactly one copy of a static variable shared by all instances of the given class.

```
static int count;
static int getCount() {...}
static void main(String[] args) {...}
```

  Class member are usually accessed using the class name.

```
Test.count = 0;
int c = Test.getCount();
```

  When they are accessed inside a given class, the class name can be omitted.

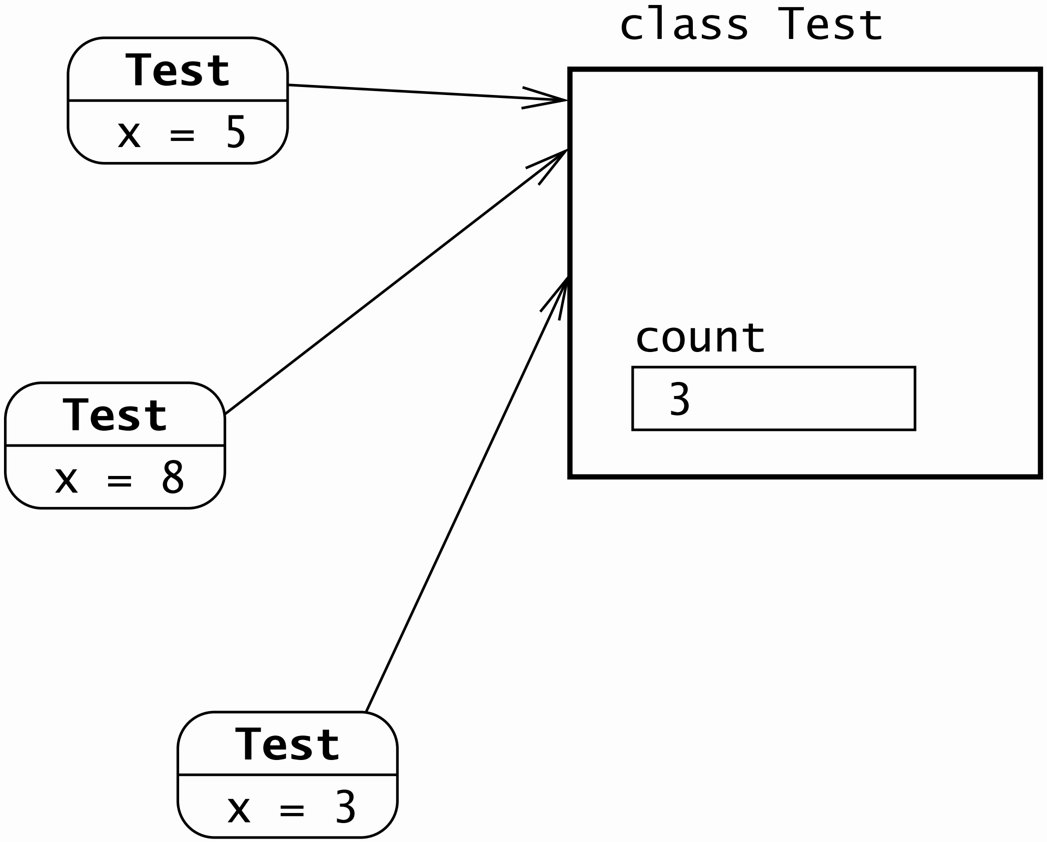
Static Members (cont.)

An example of use of static members:

```
class Test {

    private static int count = 0;
    private int x;

    public static int getCount() {
        return count;
    }

    public Test(int x) {
        this.x = x;
        count++;
    }

    public int getValue() {
        return x;
    }
}
```

Static Members (cont.)

There is exactly one copy of the static variable count in the memory. Each instance of the class Test has its own copy of the instance variable x.



Static Initializers

The static members can be initialized using **static initializer** of the form

```
static { ... }
```

The code may contain more than one static initializer. They are evaluated together with initializations of static variables in a textual order as they appear in a source file.

Static initializers are executed only once when the class is loaded into memory.

Static Initializers (cont.)

```
class StaticInitializerExample {

    static int x;

    static {
        x = 3;
        System.out.println(x);
    }

    static int y = 4;

    static {
        y = 1;
        System.out.println(y);
    }
     ...
}
```

produces the following output:

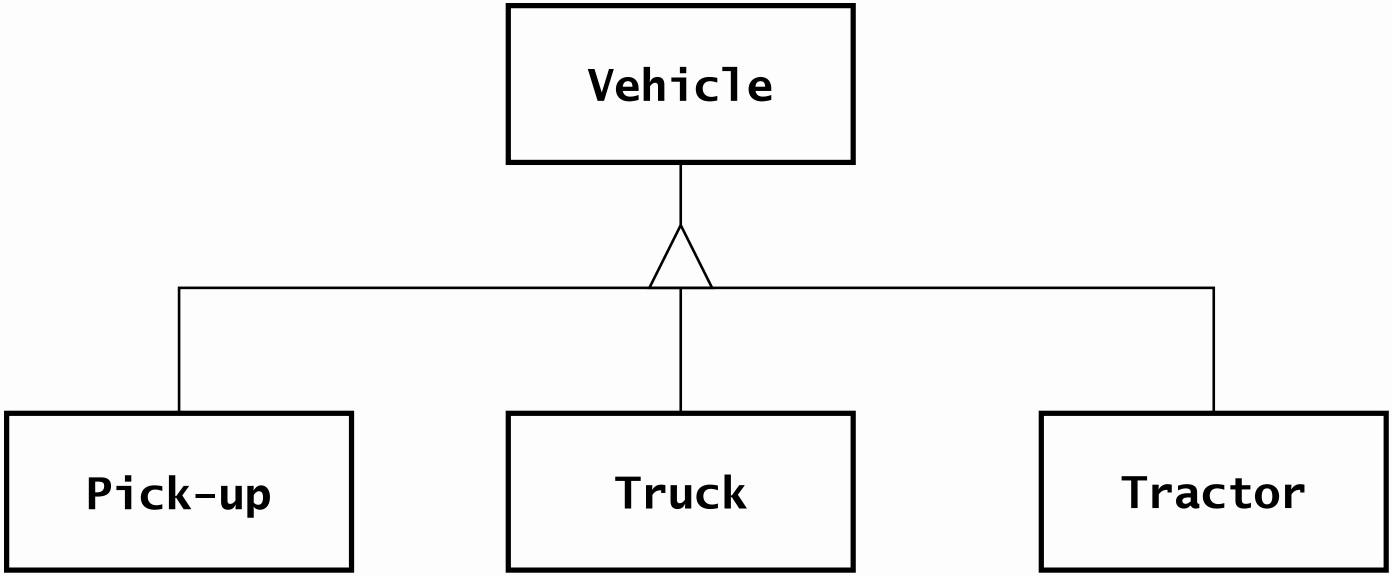
```
3
1
```

Inheritance

Object-oriented systems allow new classes to be defined in terms of a previously defined class.

All variables and methods of the previously defined class, called **superclass**, are **inherited** by **subclasses**. Subclasses can add some new variables and methods.

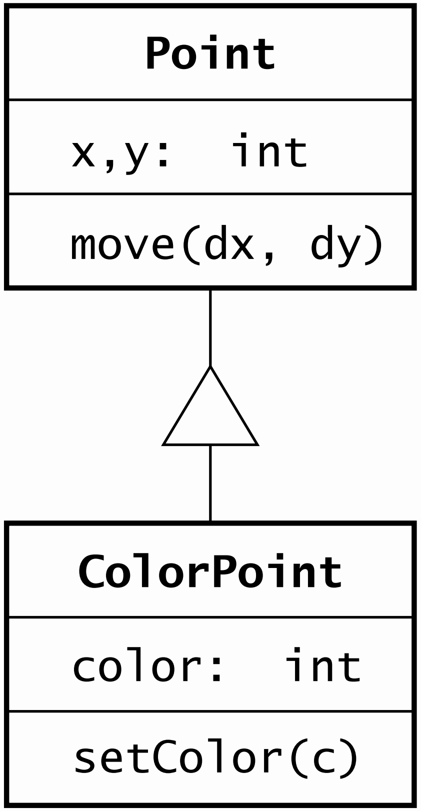
There is a hierarchical relationship between a superclass and its subclasses.



Inheritance (cont.)

Class Point represents a point in a plane.

Subclass ColorPoint adds information about color. It inherits attributes x and y and the method move() from its superclass Point.



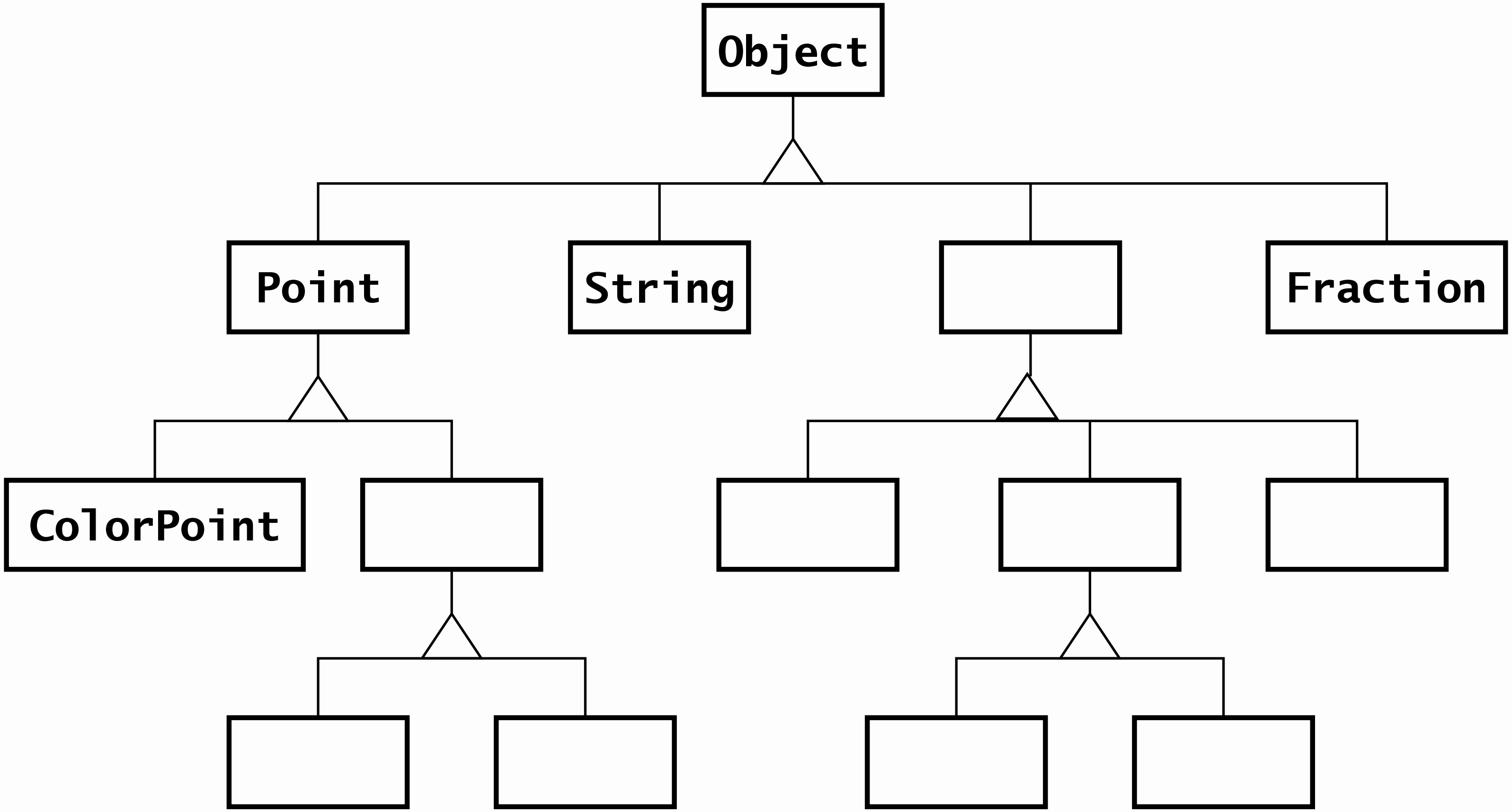
Inheritance (cont.)

Java supports inheritance through the **extends** keyword. Only single inheritance is supported, i.e. a subclass can be inherited from exactly one superclass.

```
class Point {
    private int x, y;
      ...

    public void move(int dx, int dy) {
        x += dx; y += dy;
    }
}

class ColorPoint extends Point {
    private int color;

    public void setColor(int c) {
        color = c;
    }
}
```

Hierarchy of Classes

Inheritance gives rise to a whole hierarchy of classes, because other subclasses can be inherited from subclasses of a class. Every class is a subclass of the special class Object.



Use of Subclasses

Subclasses can be used as any other classes. Attributes and methods can be accessed as usual:

```
ColorPoint p = new ColorPoint();
p.setColor(3);
p.x = 45;
p.move(10, 20);
```

Reference to an instance of a class can also point to an instance of its subclass. For example a reference to the class `Point` can point to an instance of its subclass `ColorPoint`:

```
ColorPoint p = new ColorPoint();
Point q = p;
q.move(60, -40);
Point r = new ColorPoint();
```

Use of Subclasses (cont.)

However only attributes and methods declared in the class can be accessed using reference of the given type. Attributes and methods declared in its subclasses can not be accessed using this reference.

```
Point r = new ColorPoint();
r.setColor(10);  // Compile error! Method setColor()
                 // is not defined in the class Point.
```

An instance of a subclass of a class can be assigned to a reference to the given class. On the other hand, it is a compile-time error to assign to a reference to some class an expression of type reference to its superclass:

```
ColorPoint c = new ColorPoint();
Point q = c;        //  O.K.
ColorPoint t = q;   //  Compile error!
```

Note: Every instance of ColorPoint is also an instance of Point. There can be instances of Point that are not instances of ColorPoint.

Cast Operator

It is possible to use cast operator to convert a reference to some class to a reference to its subclass:

```
Point q = new ColorPoint();
ColorPoint t = (ColorPoint)q;  // Both q and t point to
                               // the same object.
t.setColor(4);   // O.K.
q.setColor(4);   // Compile error!
```

The following usage is also possible:

```
((ColorPoint)q).setColor(4);
```

When the instance is not an instance of the class used in the cast operator, a run-time error occurs (an exception is thrown).

```
Point q = new Point();
ColorPoint t = (ColorPoint)q;  // Run-time error occurs.
```

The `instanceof` Operator

The **`instanceof`** operator determines whether a given object is an instance of particular class or type.

The syntax is:

```
    expr instanceof type
```

where `expr` represents an expression that evaluates to a reference and `type` is a name of a class.

The result of the **`instanceof`** operator is **`true`** if the value of `expr` is not **`null`** and could be cast to the `type` without raising an exception. Otherwise the result is **`false`**.

```
Point p;
  ...
if (p instanceof ColorPoint) {
    ((ColorPoint)p).setColor(5);
}
```

Note: If it is clear at compile-time that the value of the expression can not be an instance of the given class, a compile error is produced.

The final Classes

When a class is marked as **final** no subclasses can be inherited from this class.

In the following example, we can not declare the class ColorPoint as a subclass of the class Point, since the class Point is final.

```
final class Point
{
        ...
}

class ColorPoint extends Point    // Compile error!
{
        ...
}
```

Polymorphism

A subclass can **override** methods of its superclass, i.e., it can provide its own implementation of these methods.

```
class Point {
      ...
    public void print() {
        System.out.print("(" + x + "," + y + ")");
    }
}

class ColorPoint extends Point {
      ...
    public void print() {
        System.out.print("(" + x + "," + y + ", color=" +
                         color + ")");

    }
}
```

Polymorphism (cont.)

If an overridden method is called, the method in the subclass is always used. The overridden method in the superclass is not accessible from other objects.

```
ColorPoint c = new ColorPoint();
Point p = c;
c.print();  // The method print() defined in the class
p.print();  // ColorPoint is called in both cases.
```

The code that calls overridden methods does not need to be aware of different implementations of the methods in different subclasses.

```
Point[] points = new Point[10];
points[0] = new ColorPoint();
points[1] = new Point();
  ...
for (int i = 0; i < points.length; i++) {
    points[i].print();
}
```

The super Keyword

The **super** keyword can be used to access members of a class inherited by the class in which it appears.

In particular it is the only way to access overridden methods.

```
class Point {
      ...
    public void print() {
        System.out.print("(" + x + "," + y + ")");
    }
}

class ColorPoint extends Point {
      ...
    public void print() {
        super.print();  // calls the method print() in
                        // the class Point
        System.out.print(", color=" + color);
    }
}
```

The `final` Methods

A method can be marked as **`final`**. Such methods can not be overridden in subclasses.

```
class Point {
      ...
    public final void print() {  // Marked as 'final'.
        System.out.print("(" + x + "," + y + ")");
    }
}

class ColorPoint extends Point {
      ...
    public void print() {  // Compile error! Can not override
                           // final method.
        System.out.print("(" + x + "," + y + ", color=" +
                         color + ")");

    }
}
```

Inheritance and Constructors

Constructors are not inherited. The subclass must define its own constructors.

The constructors in the subclass can call a constructor of the superclass using the keyword **super**.

```
class Point {
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
      ...
}

class ColorPoint extends Point {
    public ColorPoint(int x, int y, int c) {
        super(x, y);  // The constructor in the class
                      //  Point is called.
        color = c;
    }
      ...
}
```

Inheritance and Constructors (cont.)

When no constructor of the superclass is called explicitly in the constructors in the subclass, the constructor with no parameters is used, as if the following construction would be put at the beginning of the constructor:

```
{
     super();
        ...
}
```

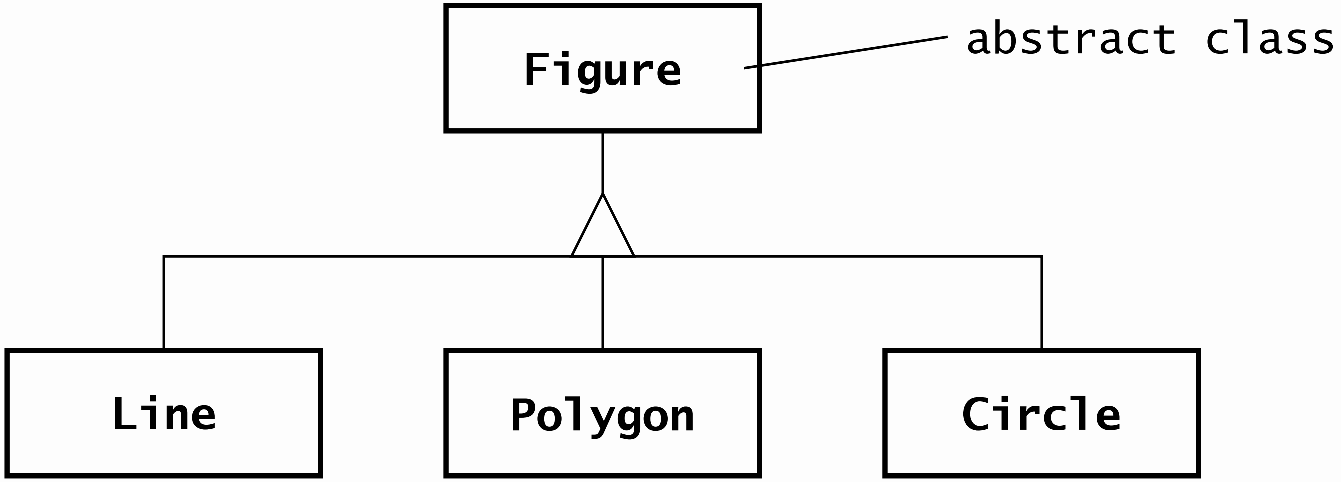
It is an compile-time error if the superclass does not define (either implicitly or explicitly) the constructor with no parameters in this case.

Abstract Classes

An **abstract** class is an incomplete description of something; a set of operations and attributes that, in themselves, do not fully describe an object.

Abstract classes are used as common superclasses of some classes and they contain common attributes and methods of these classes.

Abstract classes can not be instantiated, but their non-abstract subclasses can.



Abstract Classes (cont.)

Abstract classes are declared with the keyword **abstract**.

```
abstract class Figure {      // an abstract class
    ...
}

class Line extends Figure {  // a non-abstract class
    ...
}
```

It is possible to use references to instances of an abstract class.

```
Figure a = new Line();
a.move(10, 20);
```

It is not possible to create instances of an abstract class.

```
Figure b = new Figure();  // Compile error!
```

Abstract Methods

Abstract classes can contain abstract methods. Such methods are marked with the keyword **abstract** and have only header, their body is replaced with a semicolon (;).

```
abstract class Figure {
       ...
    abstract void draw();  // abstract method
}
```

Every abstract method must be **implemented** in non-abstract subclasses.

```
class Line extends Figure {
       ...
    void draw() {  // implementation of the abstract method
        ...  // <- draws the line
    }
}
```

Note: Every class containing a non-implemented abstract method (either directly or inherited) must be declared as an abstract class.

Abstract Methods (cont.)

Abstract methods are called as any other methods – the implementation in the corresponding subclass is called.

```
Figure[] figures = new Figure[100];
figures[0] = new Line();
figures[1] = new Circle();
figures[2] = new Polygon();
   ...
for (int i = 0; i < figures.length; i++) {
    figures[i].draw();  // The method draw() of the
                        // corresponding class is called.
}
```

Interfaces

An **interface** is a named collection of method definitions (without implementations). An interface can also declare constants.

A definition of an interface resembles a definition of a class, but the keyword **interface** is used instead of the keyword **class**.

```
interface Drawable
{
    void draw();                    // methods
    void highlight(int mode);

    int HM_DARK  = 0;               // constants for
    int HM_LIGHT = 1;               // highlight mode
}
```

The definitions of methods must be the same as definitions of abstract methods except that the keyword **abstract** is not used.

Interfaces (cont.)

We say a class **implements** an interface if it provides implementations of methods in the interface (in the same way as it implements abstract methods).

The used syntax is illustrated in the following example.

```
class Line implements Drawable
{
      ...
    public void draw() {
        ...          // a method that actually draws the line
    }

    public void highlight(int mode) {
        ...          // a method that actually highlights
                     // the line using the specified mode
    }
}
```

Interfaces (cont.)

A references that point to any object implementing the given interface can be used in the same way as references pointing to class instances.

```
Line l = new Line();
   ...
Drawable d = l;
d.draw();                         // O.K.
d.highlight(Drawable.HM_DARK);    // O.K.
d.move(10, 20);                   // Compile error. The method
                                  // move() is not deklared in
                                  // the interface Drawable.
```

Only methods declared in the interface can be called using a reference type corresponding to this interface.

Interfaces (cont.)

- An **interface** defines a protocol of behavior that can be implemented by any class **anywhere** in the class hierarchy.
- An interface **declares** a set of methods but **does not** implement them.
- A class that **implements** the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.
- There is a hierarchy of interfaces similar to hierarchy of classes. We talk about **superinterfaces** and **subinterfaces**.

```
interface DrawableFull extends Drawable
{
    void fill(int color);
}
```

Interfaces (cont.)

- A class can implement more than one interface. Names of multiple interfaces are separated by comma (`,`).

```
      class Polygon extends Figure implements
                                   Drawable, Rotating {
            ...
```

- Methods declared in an interface are implicitly **`public`** and **`abstract`**. It is not possible to change this.
- Attributes declared in an interface are implicitly **`public`**, **`static`** and **`final`**, i.e., they represent constants. It is not possible to change this.
- When a class implements an interface, it is essentially signing a contract. Either the class must implement **all** the methods declared in the interface and its superinterfaces, or the class must be declared **abstract**.

Interfaces (cont.)

The most significant differences between interfaces and abstract classes:

- An interface cannot implement any methods, whereas an abstract class can.
- An interface cannot declare any static methods, whereas an abstract class can.
- An interface cannot declare instance variables, whereas an abstract class can.
- An interface cannot declare non-final static attributes, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy - unrelated classes can implement the same interface.

The `final` Attributes

Constant values can be declared using the keyword **`final`**.

```
final int NUMBER = 10;
```

We can assign a value to **`final`** attributes and (local) variables only in their declarations or in constructors. An attempt to assign them a value in normal methods results in a compile-time error.

```
NUMBER = 5;  //  Compile error!
```

A **`final`** attribute is usually declared as **`static`**, since it is not necessary to have a copy of the same value in all instances, and one common copy is sufficient.

```
static final int NUMBER = 10;
```

Note: Names of constant values are by convention formed from upper-case letters and underscores (_).

The final Attributes (cont.)

One common usage of **final** attributes is to use them for representation of possible values from some finite set of values – **enumeration** of these values. To each possible element of the set we assign some arbitrary integer value. In program we always use the assigned symbolic names instead of integer values.

In this case names of attributes representing values from the set share a common prefix.

For example in a chess-playing program we can represent different pieces using the following declarations.

```
// chess pieces
public static final int P_NONE   = 0,
                        P_KING   = 1,
                        P_QUEEN  = 2,
                        P_BISHOP = 3,
                        P_KNIGHT = 4,
                        P_ROOK   = 5,
                        P_PAWN   = 6;
```

Packages

To make classes easier to find and to use, to avoid naming conflicts, and to control access, programmers bundle groups of related classes and interfaces into **packages**.

A **package** is a program module that contains classes, interfaces and other packages (**subpackages**).

Each package has a **name**:

- a single identifier – a name of a top level package
- of the form `Q.Id`, where `Q` is a name of a package and `Id` is an identifier – a name of a subpackage

Examples of package names:

```
points
java.lang
com.sun.security
drawing.figures
```

Packages (cont.)

A package to which a class or an interface belongs is specified at the beginning of the source file containing this class or interface using the following syntax, where name is the name of the package:

```
package name;
```

For example, a file `Line.java` may look like this:

```
package drawing.figures;

class Line extends Figure {
      ...
}
```

This specifies the class `Line` that belongs to the package `drawing.figures`.

When no package is specified at the beginning of the source file, the class or interface defined in this file belongs to a special **unnamed package**.

Canonical Names

Each class or interface has a **fully qualified (canonical) name** that specifies also the package to which it belongs.

For example, the fully qualified name of the previously defined class `Line` is:

```
drawing.figures.Line
```

It is possible to use the same name for two classes or interfaces as long as they belong to different packages (and so they have different canonical names).

For example, can define another class `Line` in a package `net.connections` with the canonical name:

```
net.connections.Line
```

The canonical name of a class or interface that belongs to the unnamed package is the name of this class or interface, for example:

```
Line
```

Canonical Names (cont.)

We can always use a fully qualified name of a class or interface when we refer to this class or interface:

```
drawing.figures.Line line = new drawing.figures.Line();
```

When we use a simple name (i.e., when we do not use the canonical name) we refer to a class or interface in the current package:

```
Line line = new Line();
```

A package may not contain two members of the same name, or a compile-time error results. For example:

- The package `drawing.figures` cannot contain other class or interface named `Line`.
- The package `drawing.figures` cannot contain a subpackage `Line`.
- The package `drawing` cannot contain a class or interface named `figures`.

Public Classes and Interfaces

Only classes or interfaces declared **`public`** can be accessed in other packages.

For example, the class `drawing.figures.Line` declared this way can be accessed in all packages:

```java
package drawing.figures;

public class Line extends Figure {
      ...
}
```

If it would be declared the following way then it can be accessed only in (classes and interfaces in) the package `drawing.figures`:

```java
package drawing.figures;

class Line extends Figure {
      ...
}
```

Hierarchy of Packages

The hierarchical naming structure for packages is intended to be convenient for organizing related packages, but has no other significance.

For example there is no special access relationship between classes defined in the following packages:

```
drawing.figures
drawing.figures.colors
drawing.menu
```

Package names correspond to directories in a file system. For example the class `drawing.figures.Line` should be stored in a file named

```
drawing/figures/Line.java     (on Unix)
```

resp.

```
drawing\figures\Line.java     (on MS Windows)
```

Note: Files containing classes from the unnamed package should be stored in the current working directory.

Import Declarations

It is always possible to refer to classes and interfaces from other packages using their canonical names.

It is possible to use import declarations to import classes and interfaces from other packages and to refer to them using simple names.

The are two types of import declarations:

- single type declarations – imports one class or interfaces

```
import drawing.figures.Line;
```

- import on demand declarations – imports all public classes and interfaces from the given package

```
import drawing.figures.*;
```

A source file can contain any number of import declarations.

Import Declarations (cont.)

The file drawing/figures/Line.java contains:

```
package drawing.figures;

public class Line extends Figure {
      ...
}
```

The file drawing/menu/Commands.java contains:

```
package drawing.menu;

import drawing.figures.Line;   // single type import

class Commands {
    public Line createLine() {   // Line refers to the class
        Line line = new Line();  // drawing.figures.Line
           ...
    }
       ...
}
```

Import Declarations (cont.)

The file drawing/figures/Line.java contains:

```
package drawing.figures;

public class Line extends Figure {
      ...
}
```

The file drawing/menu/Commands.java contains:

```
package drawing.menu;

import drawing.figures.*;   // on demand import

class Commands {
    public Line createLine() {   // Line refers to the class
        Line line = new Line();  // drawing.figures.Line
           ...
    }
       ...
}
```

Import Declarations (cont.)

When a source file contains a single class (or interface) name then the definition of the class is found using the following procedure:

- If the class is defined in the source file or if it is imported using single type import declaration, the corresponding definition is used.
- If the class with the given name is defined in the same package (but in other file), that definition is used.
- If the class is defined in some package imported using on demand import declaration, that definition is used.
- Otherwise a compile-time error results.

Note: Only public classes and interfaces can be imported from other packages.

Import Declarations (cont.)

Some remarks:

- When a class defined in a source file has the same single name as the class imported using **single type** import declaration, a compile-time error results.

  So the following program causes a compile time error.

```
package drawing.menu;

import drawing.figures.Line;

class Line {  // Compile error!
       ...
}
```

- It is also not possible to use more than one **single type** import declaration with the same single name of a class or interface:

```
import drawing.figures.Line;
import net.connections.Line;  // Compile error!
```

Import Declarations (cont.)

- When a source file contains single type import declaration and there is a class with the same single name defined in the same package, but in other file, then the class specified in the import declaration is used.
- It is a compile-time error when a single name is used to refer to a class that is defined in two or more packages imported using on demand import declarations:

```
import drawing.figures.*; // contains class Line
import net.connections.*; // contains class Line
  ...
Line line = new Line();   // Compile error!
                          // Fully qualified
                          // name must be used.
```

- On demand import declarations do not conflict with single type import declarations or with classes defined in the given package.

Source Files

A Java source file (also called **compilation unit**) has the following structure:

- package declaration
- import declarations
- type declarations

The ordering of these parts is mandatory. Each of them is optional.

- **Package declaration** is always of the form

  ```
  package name;
  ```

  where name is the (canonical) name of the package.
- **Import declarations** is a sequence of any number of import declarations (single type and on demand).
- **Type declarations** is a sequence of any number of class and interface definitions.

Source Files (cont.)

- When a source file contains a public class or interface named X, the name of the source file must be `X.java`.
- When a source file contains a class or interface named X that is referred from other source files, the name of the source file must be `X.java`
- The above rules mean that a file may contain at most one class or interface that is either public or that is referred from other source files.
- When source files are compiled, a file `X.class` is created for each class or interface named X.
- Names of directories must correspond to names of packages.

Ant

When we have a program consisting of many source files, we usually do not run compiler manually, but use some special tool for it.
A standard tool used for this purpose for Java programs is called **Ant**.

- Dependencies between files can be specified using a special language.
- It is not always necessary to compile all source files, but only some of them. Ant automatically figures out which files should be compiled and calls compiler on them.
- Ant is not a part of JDK, but can be downloaded from `http://ant.apache.org`.
- Most of development environments use Ant internally for management of projects.

Package Names

Packages **java** and **javax** and their subpackages are reserved for standard classes, so no classes or interfaces should be defined by a user in these packages.

A short overview of the most important standard packages:

- **java.lang** – fundamental classes for Java programming language
- **java.util** – miscellaneous utility classes (abstract data types, manipulation with date and time, ...)
- **java.io** – classes for input and output from and to files and for manipulation with files
- **java.net** – classes for network communication
- **java.awt** – Abstract Window Toolkit – classes for creating user interfaces and for painting graphics and images
- **java.applet** – classes for creating applets
- **javax.swing** – modern user interface Swing
- **javax.sound** – classes for working with sound

Package Names (cont.)

Some remarks:

- Classes and interfaces from package **java.lang** are always automatically imported as if the following import declaration would be used:

  ```
  import java.lang.*;
  ```

- By convention package names contain only lower-case letters.

Package Names (cont.)

When some packages are widely distributed the following convention is suggested:

- We create a unique package name from an Internet domain name belonging to an organization that produces the package by reversing this domain name component by component.

  For example from a domain name

  ```
  mycompany.com
  ```

  we create a package name

  ```
  com.mycompany
  ```

- All other packages are then created as subpackages of this package.

This convention allows to avoid package names conflicts.

Access Control

Access to a member of a class can be specified using one of the keywords **public**, **protected** or **private**, or it may not be specified (**default access**):

- **public** – it can be accessed from any class
- **protected** – it can be accessed from any subclass and from any class in the same package
- **(default)** – it can be accessed from any class in the same package
- **private** – it can be accessed only from the class where it is defined.

When we override a method in a subclass it must be declared with the same or more permissive access than the method in the superclass.

Classes Without Instances

The standard way how to define a class such that it is not possible to create instances of this class is:

- to declare a constructor of no arguments and make it **private**
- never invoke this constructor
- declare no other constructors

Class of this form usually contains class methods and variables.

An example of such class is the class `java.lang.Math` containing standard mathematical functions:

```
public final class Math {

    private Math() { }   // never instantiate this class

         ...      // class variables and methods
}
```

Modifiers

A **modifier** specifies some special property of a class, an interface, a method, an attribute, or a constructor.

All possible modifiers in Java are:

public **protected** **private**
static **abstract** **final**
native **synchronized** **transient**
volatile **strictfp**

- Modifiers are used in front of a declaration.
- The ordering of modifiers is not important when more than one modifier is used.
- Some modifiers can be used only in some contexts.
- At most one of keywords **`public`**, **`protected`** and **`private`** can be used in one declaration.
- It is an error to use the same modifier more than once.

Modifiers (cont.)

- Class modifiers:
  public abstract final strictfp
- Field (attribute) modifiers:
  public protected private
  static final transient volatile
- Method modifiers:
  public protected private abstract static
  final synchronized native strictfp
- Constructor modifiers:
  public protected private
- Interface modifiers:
  public

Class `java.lang.Object`

Class `java.lang.Object` is a common superclass of all classes.

All objects, including arrays, inherit methods of this class:

- `equals(Object obj)` – tests if two objects are equal
- `clone()` – creates a copy of this object
- `toString()` – returns a string representation of the object
- `hashCode()` – returns a hash code value for the object
- `getClass()` – returns information about the class of the object
- `finalize()` – called by the garbage collector before the memory is freed
- `wait()`, `notify()`, `notifyAll()` – for synchronization of threads

References

There are three reference types in Java:

- class references
- interface references
- array references

There are two types of objects in Java:

- class instances
- arrays

References are **pointers** to objects. They can have **`null`** value and there can be many references to the same object.

The class `java.lang.Object` is a superclass of all other class. A variable of type `Object` can hold a reference to an instance of a class or to an array.

```
int[] a = { 3, 1, 5 };
Object o = a;
Object p = new Object[10];
```

References (cont.)

It is possible to compare two references using operators == and !=:

- The result of == is **true** if both references point to the same object or if they are both **null**.
- The result of == is **false** otherwise.
- The operator != works as a negation of ==.
- A compile-time error occurs if it is impossible to convert the type of either operand to the type of the other by a casting conversion.

```
Point a = new Point(10, 20);
Point b = new Point(10, 20);
System.out.println(a == b);   // prints 'false'
Object c = a;
System.out.println(a == c);   // prints 'true'
```

Method `equals()`

The method

```
public boolean equals(Object obj)
```

defined in the class `java.lang.Object` can be used to compare two different object if they are the same.
This method can be overridden in subclasses. If it is not overridden it behaves as if the test

```
(this == obj)
```

was used.

The method `equals()` implements an equivalence relation:

- **reflexive** – `x.equals(x)` should return **`true`**,
- **symmetric** – if `x.equals(y)` then also `y.equals(x)`,
- **transitive** – if `x.equals(y)` and `y.equals(z)` then also `x.equals(z)`,
- **consistent** – `x.equals(y)` should return always the same value, if objects pointed to by `x` and `y` has not changed,
- `x.equals(`**`null`**`)` should return **`false`**.

Method equals() (cont.)

```
class Point {
    private int x, y;

    public boolean equals(Object obj) {
        if (!(obj instanceof Point)) return false;
        Point p = (Point)obj;
        return (x == p.x && y == p.y);
    }

    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
}
```

```
Point a = new Point(10, 20);
Point b = new Point(10, 20);
Point c = new Point(30, 20);
System.out.println(a == b);       // prints 'false'
System.out.println(a.equals(b));  // prints 'true'
System.out.println(a.equals(c));  // prints 'false'
```

Copying Objects

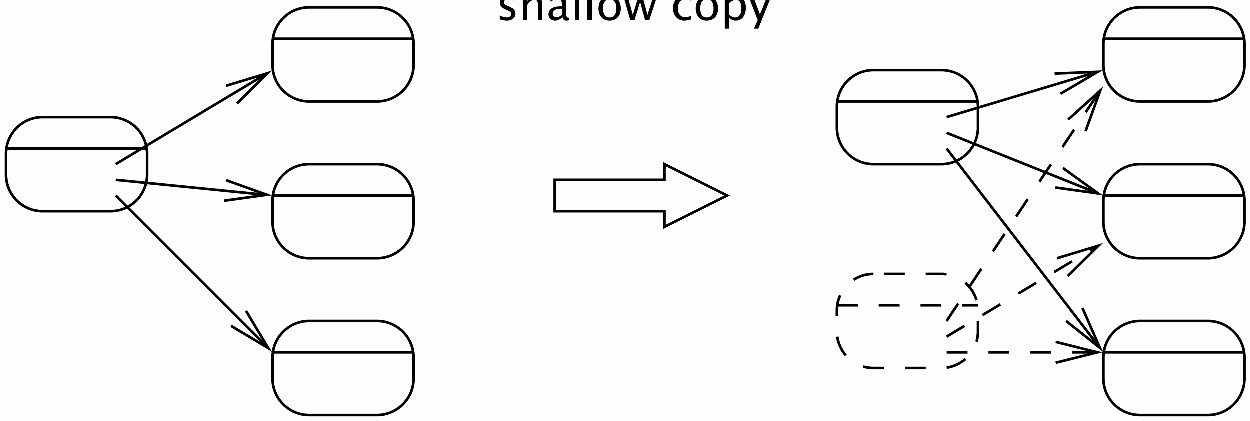
The `clone()` method is intended for creation of a copy of an object. The simplest way to make your class cloneable, is to add **`implements`** `Cloneable` to class's declaration. For some classes the default behavior of `Object`'s `clone()` method works just fine. Other classes need to override clone to get correct behavior.

- **Shallow Copy**
  A clone of an original object is created only. All instance variables of the clone have the same values as the ones of the original, i.e. if a variable holds reference to an object, the original and the copy refer to the same object.
- **Deep Copy**
  Copies of an original object and all its instance variables are created. Then, any modification of the original does not affect its copy and vice versa.

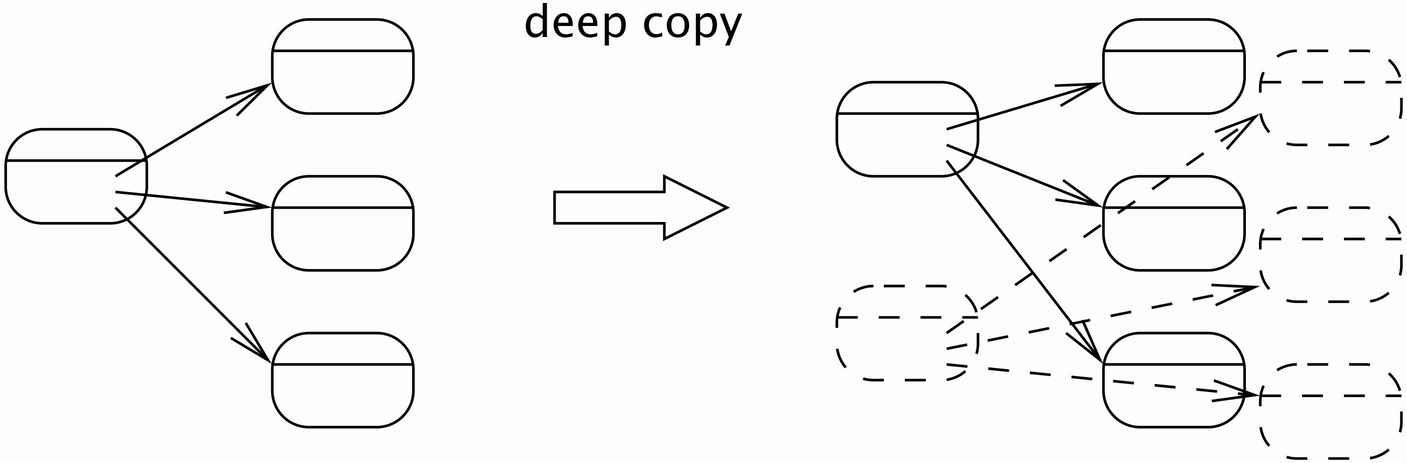
Note: The `clone()` should never use **new** to create the clone and should not call constructors. Instead, the method should call **`super.`**`clone()`, which creates an object of the correct type and allows the hierarchy of superclasses to perform the copying necessary to get a proper clone.

Copying Objects (cont.)

shallow copy



deep copy



Garbage Collection

When garbage collector is ready to release a memory used for an object, it will first call `finalize()` method, and only then the memory is reclaimed. Usage of `finalize()` gives the ability to perform some important cleanup **at the time of garbage collection**.

```
protected void finalize()
```

If there is some activity that must be performed before an objects is no longer need, the activity must be performed by programmer. Java has no destructor or similar concept, so an ordinary method performing this cleanup must be created.

> **Note:** It is not good idea to rely on `finalize()` being called, and separate “cleanup” functions should be created and called explicitly.

Garbage collection can be characterized as follows:

- garbage collection is not destruction,
- some objects might not get garbage-collected,
- garbage collection is only about memory.

Strings

Strings are sequences of characters (primitive type **char**).

There are two classes in Java that can be used to represent strings (both are from the package `java.lang`):

- **String**
- **StringBuffer**

All string literals, such as "abc", are represented as instances of the class **String**.

Strings are constants, their values cannot be changed after they are created.

The class **StringBuffer** supports mutable strings.

Strings in Java are not arrays of characters. This means that **char**`[]` is not `String` and vice versa.

However, character arrays can be used when we work with strings. Also both classes **String** and **StringBuffer** use character arrays in their internal implementation.

Class String

The easiest way how strings can be created is to use string literals:

```
String s = "abc";
```

Notice that no operator **new** is used in this case. The object of the class String is created automatically in this case.

The class String has many different constructors:

- String()
- String(String original)
- String(StringBuffer buffer)
- String(**char**[] value)
- String(**char**[] value, **int** offset, **int** count)
- String(**byte**[] bytes, String charsetName)
- ...

Class String (cont.)

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

char data2[] = {'a', 'b', 'c', 'd', 'e', 'f'};
String t = new String(data2, 2, 3);  //  t = "cde";
```

The most important methods in the class `String`:

- **`int`** `length()` – returns the length of the string
- **`char`** `charAt(`**`int`** `index)` – returns the character at the specified index
- **`boolean`** `equals(Object obj)` – compares two strings

```
String s = "abcdef";
System.out.println(s.length());          // prints '6'
System.out.println(s.charAt(5));         // prints 'f'
System.out.println(s.equals("abcdef"));  // prints 'true'
System.out.println(s.equals("hello"));   // prints 'false'
```

Class String (cont.)

Methods that transform strings to arrays:

- void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
- char[] toCharArray()
- byte[] getBytes(String charsetName)
- byte[] getBytes()

Methods for comparison of strings:

- boolean equals(Object obj)
- boolean equalsIgnoreCase(String str)
- int compareTo(String str)
- int compareToIgnoreCase(String str)
- boolean contentEquals(StringBuffer sb)

Note: Methods `compareTo()` and `compareToIgnoreCase()` return a negative integer, zero, a positive integer if the the specified string is greater than, equal to, or less than this string.

Class String (cont.)

Methods for searching for a character in a string:

- int indexOf(int ch)
- int indexOf(int ch, int fromIndex)
- int lastIndexOf(int ch)
- int lastIndexOf(int ch, int fromIndex)

Methods for searching for a substring in a string:

- int indexOf(String str)
- int indexOf(String str, int fromIndex)
- int lastIndexOf(String str)
- int lastIndexOf(String str, int fromIndex)

Note: The value -1 is returned if the searched character or substring is not found.

Class String (cont.)

The methods for obtaining substrings:

- `String substring(int beginIndex)`
- `String substring(int beginIndex, int endIndex)`

```
String s = "abcdef";
System.out.println(s.substring(2,5)); // prints 'cde'
```

The methods for comparison of substrings:

- boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
- boolean startsWith(String prefix, int offset)
- boolean startsWith(String prefix)
- boolean endsWith(String suffix)

Class `String` (cont.)

Other methods that manipulate strings:

- String toLowerCase()
- String toUpperCase()
- String concat(String str) – concatenates the specified string to the end of this string
- String replace(char oldChar, char newChar) – replaces all occurrences of `oldChar` with newChar
- String trim() – removes leading and trailing whitespace

There are also static methods called `valueOf()` that transform different types of values to strings:

- static String valueOf(boolean b)
- static String valueOf(char c)
- static String valueOf(int i)
- . . .

Strings – Example

```
public class Filename {
    private String fullPath;
    private char pathSeparator, extensionSeparator;

    public Filename(String str, char sep, char ext) {
        fullPath = str;
        pathSeparator = sep;
        extensionSeparator = ext;
    }

    public String getExtension() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        return fullPath.substring(dot + 1);
    }

    public String getFilename() {
        int dot = fullPath.lastIndexOf(extensionSeparator);
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(sep + 1, dot);
    }
```

Strings – Example

```
    public String getPath() {
        int sep = fullPath.lastIndexOf(pathSeparator);
        return fullPath.substring(0, sep);
    }
}
```

The following code illustrates usage of `Filename`:

```
Filename myHomePage = new Filename("/home/mem/index.html",
                                   '/', '.');
System.out.println("Extension = " + myHomePage.getExtension());
System.out.println("Filename = " + myHomePage.getFilename());
System.out.println("Path = " + myHomePage.getPath());
```

Produced output:

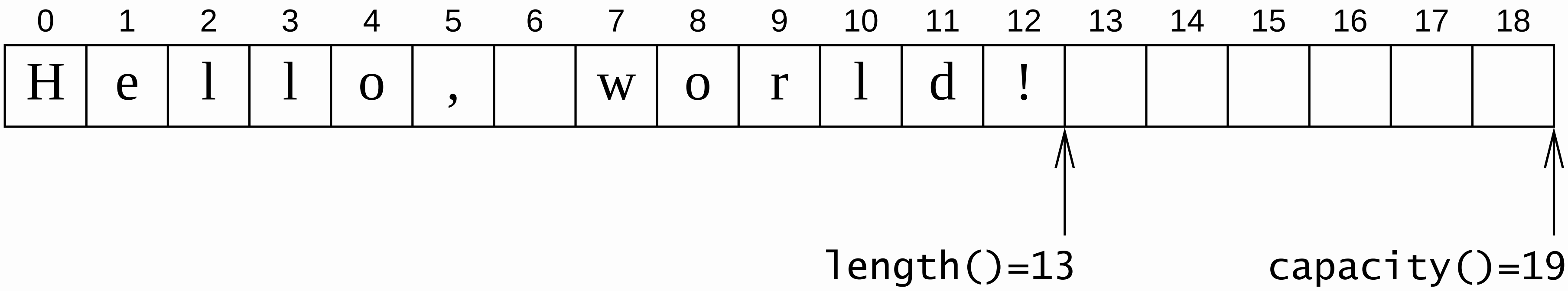
```
Extension = html
Filename = index
Path = /home/mem
```

Class StringBuffer

The class `StringBuffer` implements a mutable sequence of characters, the length and content of the sequence can be changed through certain method calls.

The most important methods:

- int length()
- void setLength(int newLength)
- char charAt(int index)
- void setCharAt(int index, char ch)
- int capacity()
- void ensureCapacity(int minimumCapacity)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
H	e	l	l	o	,		w	o	r	l	d	!						

`length()=13`

`capacity()=19`

Class StringBuffer (cont.)

The constructors:

- StringBuffer()
- StringBuffer(int length)
- StringBuffer(String str)

The most important methods used to modify a string buffer are:

- **append** – adds characters at the end of the buffer,
- **insert** – adds characters at a specified point.

```
StringBuffer b = new StringBuffer("abcd");
b.append("ef");       // 'b' contains 'abcdef'
b.insert(3, "ghi");   // 'b' contains 'abcghidef'
```

The contents of a string buffer can be transformed into a string using the `toString()` method:

```
String s = b.toString();   // 's' contains 'abcghidef'
```

Class StringBuffer (cont.)

Methods append and insert are overloaded so as to accept data of any type:

- StringBuffer append(String str)
- StringBuffer append(StringBuffer sb)
- StringBuffer append(Object obj)
- StringBuffer append(char[] str)
- StringBuffer append(boolean b)
- StringBuffer append(char c)
- StringBuffer append(int i)
- StringBuffer append(double d)
- . . .
- StringBuffer insert(int offset, String str)
- StringBuffer insert(int offset, char[] str)
- . . .

Class StringBuffer (cont.)

It is also possible to delete characters:

- StringBuffer delete(int start, int end)
- StringBuffer deleteCharAt(int index)

Other methods that manipulate string buffers:

- StringBuffer replace(int start, int end, String str)
- String substring(int start)
- String substring(int start, int end)

The methods for searching in string buffers:

- int indexOf(String str)
- int indexOf(String str, int fromIndex)
- int lastIndexOf(String str)
- int lastIndexOf(String str, int fromIndex)

Operator +

Operator + can be used for concatenation of strings:

```
String a = "Hello";
String b = ", world";
String c = a + b;  // c = "Hello, world"
```

It is also possible to use +=:

```
String a = "Hello";
a += ", world";   ;  // a = "Hello, world"
```

If at least one of operands of + is a string, the other operand is transformed into a string automatically:

```
String a = "Result=" + (3 + 2) ;  // a = "Result=5"
String b = "Result=" + 3 + 2 ;    // b = "Result=32"
```

Operator + (cont.)

String buffers are used by the compiler to implement the string concatenation operator +. For example, the code:

```
String x;
x = "a" + 4 + "c";
```

is compiled to the equivalent of:

```
String x;
x = new StringBuffer().append("a").append(4).append("c")
                               .toString();
```

It is usually more efficient to manipulate a `StringBuffer` then to manipulate `String`. A new instance of `String` is created after every operation.

StringBuffer – Example

```java
class StringBufferDemo {
    static String toString(int[] a) {
        StringBuffer buf = new StringBuffer("{ ");
        for (int i = 0; i < a.length; i++) {
            if (i > 0) buf.append(", ");
            buf.append(a[i]);
        }
        buf.append(" }");
        return buf.toString();
    }

    public static void main(String[] args) {
        int[] a = { 81, 32, 53, 21, 82 };
        String s = toString(a);
        System.out.println(s);
    }
}
```

```
Output: { 81, 32, 53, 21, 82 }
```

Primitive Type Wrappers

Each primitive type has its object wrapper. Although usage of primitive types is more efficient, there are some situations where application of their object wrappers is either more convenient or just inevitable.

Primitive type	Size [bits]	Wrapper class
boolean	-	Boolean
char	16	Character
byte	8	Byte
short	16	Short
int	32	Integer
long	64	Long
float	32	Float
double	64	Double
void	-	Void

Instances of wrapper classes represent immutable values.

Primitive Type Wrappers (cont.)

There is a common abstract superclass of `Byte`, `Short`, `Integer`, `Long`, `Float` and `Double` called `Number`.

Wrapper classes contain also many useful static methods for manipulation with values of the given primitive type:

- methods that transform the values of the primitive type to strings
- methods that transform strings to the given primitive type

Wrapper classes for numeric types also contain static constants `MIN_VALUE` and `MAX_VALUE` representing the minimal and maximal value of the given numeric type.

```
short x = Short.MIN_VALUE;  // x = -32768
```

Class Character

Class `Character` contains many static methods for testing if a character belongs to the given category of characters:

- static boolean isLowerCase(char ch)
- static boolean isUpperCase(char ch)
- static boolean isDigit(char ch)
- static boolean isDefined(char ch)
- static boolean isLetter(char ch)
- static boolean isLetterOrDigit(char ch)
- static boolean isSpaceChar(char ch)
- static boolean isWhitespace(char ch)
- static boolean isISOControl(char ch)
- `. . .`

Java uses 16-bit character encoding called Unicode. More information about Unicode can be found at `http://www.unicode.org`.

ASCII Table

The first 128 characters of Unicode are the same as in the ASCII table.

ASCII – American Standard Code of Information Interchange

0	NUL	16	DLE	32		48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Manipulation With Characters

Some examples of manipulation with characters. Notice how properties of ASCII are used.

- Transformation of a decimal digit to the numerical value:

```
static int decToNum(char c) {
    if (c >= '0' && c <= '9') return c - '0';
    return -1;
}
```

- Transformation of a hexadecimal digit to the numerical value:

```
static int hexToNum(char c) {
    if (c >= '0' && c <= '9')
        return c - '0';
    else if (c >= 'A' && c <= 'F')
        return c - 'A' + 10;
    else if (c >= 'a' && c <= 'f')
        return c - 'a' + 10;
    return -1;
}
```

Class Character (cont.)

Class `Character` contains static methods for transformation of characters to lower-case and upper-case:

- static char toLowerCase(char ch)
- static char toUpperCase(char ch)

These methods work for all Unicode characters.

Simplified version of `toLowerCase()` working only on ASCII characters could look like this:

```
static char toLowerCase(char c) {
    if (c >= 'A' && c <= 'Z') {
        return (char)(c - 'A' + 'a');
    }
    return c;
}
```

Class Integer

Class Integer contains the following static method for transformation of integer values to strings:

- static String toString(int i, int radix)
- static String toString(int i)
- static String toOctalString(int i)
- static String toHexString(int i)
- static String toBinaryString(int i)

Class Long contains similar methods that work with type **long.**

We can use the following methods to transform a string into an integer:

- static int parseInt(String s, int radix)
- static int parseInt(String s)
- static Integer valueOf(String s, int radix)
- static Integer valueOf(String s)
- static Integer decode(String nm)

Class Number

The abstract class Number defines the following (instance) methods:

- byte byteValue()
- short shortValue()
- int intValue()
- long longValue()
- float floatValue()
- double doubleValue()

Each subclass of Number (Byte, Short, Integer, Long, Float, Double) implements these methods.

Subclasses of Number implemet static methods for transformation of strings into the corresponding primitive type:

- Class **Byte**:
  - static byte parseByte(String s, int radix)
  - static byte parseByte(String s)
- . . .

Classes Float and Double

Classes Float and Double contain static fields representing some special values of types **float** and **double**:

- NaN – Not-a-Number
- NEGATIVE_INFINITY
- POSITIVE_INFINITY

These values can be used as any other **float** or **double** values.

There are also static methods that allow to test these special values:

- static boolean isNaN(float v)
- static boolean isInfinite(float v)
- static boolean isNaN(double v)
- static boolean isInfinite(double v)

Class Math

The class **java.lang.Math** contains methods for performing basic numeric operations such as:

- exponential
- logarithm
- square root
- trigonometric functions

The class **Math** contains two important static constants:

- E – Euler number, the base of the natural logarithms (2.71828...)
- PI – number *pi* (3.14159...)

It is not possible to create instances of **Math.**

All methods are static.

Class Math (cont.)

Overview of methods:

- static double exp(double a)
- static double log(double a)
- static double sqrt(double a)
- static double pow(double a, double b)
- static double sin(double a)
- static double cos(double a)
- static double tan(double a)
- static double asin(double a)
- static double acos(double a)
- static double atan(double a)
- static double atan2(double y, double x)
- static double toRadians(double angdeg)
- static double toDegrees(double angrad)

Class Math (cont.)

Other methods:

- static double ceil(double a)
- static double floor(double a)
- static double rint(double a)
- static int round(float a)
- static long round(double a)
- static double random()
- static int abs(int a)
- static long abs(long a)
- static float abs(float a)
- static double abs(double a)
- static int max(int a, int b)
- static int min(int a, int b)
- . . .

Class Math (cont.)

Example of usage of mathematical functions:

```
double start = 0.0;
double end = Math.PI * 2.0;
double step = 0.05;
for (double x = start; x <= end; x += step) {
    double y1 = Math.sin(x);
    double y2 = Math.cos(x);
    System.out.print("x=" + x);
    System.out.print("  sin(x)=" + y1);
    System.out.println("  cos(x)=" + y2);
}
```

One possible way how numbers can be rounded (except using the method round()):

```
double a;
  ...
int i = (int)(a + 0.5);
```

Method `main()`

At least one class in a program must contain the static method

```
    public static void main(String[] args)
```

Such classes can be used to start programs:

```
  $ java MyClass
```

The method `main()` usually creates instances of other classes and calls their methods.

An argument to the method `main()` is an array of strings containing command line arguments. For example when we use

```
  $ java MyClass Hello World
```

in the method `main()` in the class `MyClass` we have

```
    args.length = 2
    args[0] = "Hello"
    args[1] = "World"
```

Program Exit

The program exits when the method `main()` is finished.

> **Note:** In fact, it is more complicated. Generally, program exits when all threads are finished.

It is also possible to use the method `exit()` in the class **java.lang.System** to exit program:

```
System.exit(0);
```

The argument of `exit()` is a a status code. By convention, a nonzero status code indicates abnormal termination.

```
System.exit(1);  // an error occured
```

Exceptions

The Java programming language provides a mechanism known as **exceptions** to help programs report and handle errors:

- When an error occurs, the program throws an exception.
- The normal flow of the program is interrupted and the runtime environment attempts to find an **exception handler**, a block of code that can handle a particular type of error.

Exception – an event that occurs during the execution of a program that disrupts the normal flow of instructions.

An object containing an information about the event is also called an exception.

Exceptions – Example

An example of usage of exceptions:

```
public static void main(String[] args) {
   try {
      int c = Integer.parseInt(args[0]);
      while (c-- > 0)  System.out.println(args[1]);
   }
   catch (ArrayIndexOutOfBoundsException e) {
      System.err.println("Missing argument");
   }
   catch (NumberFormatException e) {
      System.err.println("\"" + args[0] +
              "\" isn't an integer");
   }
}
```

```
Usage: java Example 3 xyz
```

Exceptions – Motivation

Let us consider a method that reads an entire file into memory. In pseudo-code it looks like this:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

Exceptions – Motivation (cont.)

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
            . . .
```

Exceptions – Motivation (cont.)

```
            . . .
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Exceptions – Motivation (cont.)

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Exception Objects

When an error occurs in Java:

- An exception object is created, it contains information about the exception (its type, the state of the program when the error occurred, ...).
- Normal flow of instructions is disrupted.
- The runtime system finds some code to handle the error.

An exception object is always an instance of some subclass of the class **java.lang.Throwable.** There are many standard exception classes and it is possible to define own exception classes.

Creating an exception and handing it to the runtime system is called **throwing an exception**.

The code that handles the exception is called an **exception handler**. The exception handler is said to **catch the exception**.

Which exception handler is chosen depends on the type of the exception object.

Catching Exceptions

There are three main components of a code that catches exceptions:

- the **try** block
- the **catch** blocks
- the **finally** block

The syntax is:

```
try  {
    . . .
} catch (. . .) {
    . . .
} catch (. . .) {
    . . .
} finally {
    . . .
}
```

A **try** block **must** be accompanied by at least one **catch** block or one **finally** block.

The try Block

In general a **try** block looks like this:

```
try {
    . . .   // Java statements
}
```

A **try** block is said to **govern** the statements enclosed within it and defines the scope of any exception handlers.

If an exception occurs within the **try** statement, that exception is handled by the appropriate exception handler associated with this **try** statement.

There can be any number of the **catch** blocks, but at most one **finally** block.

The catch Block(s)

The general form of a **catch** block is:

```
catch (SomeThrowableObject variableName) {
    . . .   // Java statements
}
```

A class `SomeThrowableObject` is a subclass of **java.lang.Throwable**. It declares the type of exceptions the handler can handle.

The variable `variableName` is the name by which the handler can refer to the exception.

This is a declaration of a local variable `variableName`. The scope of this variable is the body of the **catch** block.

The variable `variableName` can be used as any other local variable:

```
variableName.getMessage();
```

Note: The conventional name used for these types of variables is e.

The catch Block(s) (cont.)

The **catch** block contains a series of statements that are executed when the exception handler is invoked:

- If no exception occurs in the **try** block, all its **catch** blocks are skipped and the execution continues after them.
- If an exception of type T occurs in the **try** block and there is a **catch** block handling exceptions of type T (or its superclass), then this block is executed.

  If there is more than one handler that handles exceptions of type T then the first one matching handler is used.
- If there is no such handler, the runtime system looks for some other enclosing **try** statement and its handlers.

  Note: Exceptions can be thrown everywhere, even inside the **catch** blocks.

The catch Block(s) (cont.)

The typical use of exception handlers:

```
try {
       . . .
} catch (ArithmeticException e) {
    System.out.println("Caught ArithmeticException: " +
                       e.getMessage());
} catch (IOException e) {
    System.out.println("Caught IOException: " +
                       e.getMessage());
}
```

The `finally` Block

The **finally** block provides a mechanism that allows to clean up the state of a method **regardless** of what happens within the **try** block.

Statements in the **finally** block are performed after:

- the **try** block exited normally,
- an exception occurred in the **try** block and was caught by some exception handler,
- an exception occurred in the **try** block and was not caught.

```
try {
      . . .  // opens a file and writes to it
} finally {
    if (file != null) {
        file.close();
    }
}
```

Exceptions and Methods

A method need not catch all exceptions, it can also throw exceptions to its caller.

If an exception of type T can occur in a method and the method does not catch the exception of type T, then we must specify that the method can throw an exception of type T.

To specify this, we add a **throws** clause to the header of the method:

```
public void readFile(String filename) throws IOException
{
       . . .
}
```

If a method can throw more than one type of exception we must specify all of them:

```
public Connection openConnection(Address addr)
    throws ConnectException, UnknownAddrException {
       . . .
```

Exceptions and Methods (cont.)

Any exception that can be thrown by a method is part of the method's public programming interface: callers of a method must know about the exceptions that a method can throw to intelligently and consciously what to do about those exceptions.

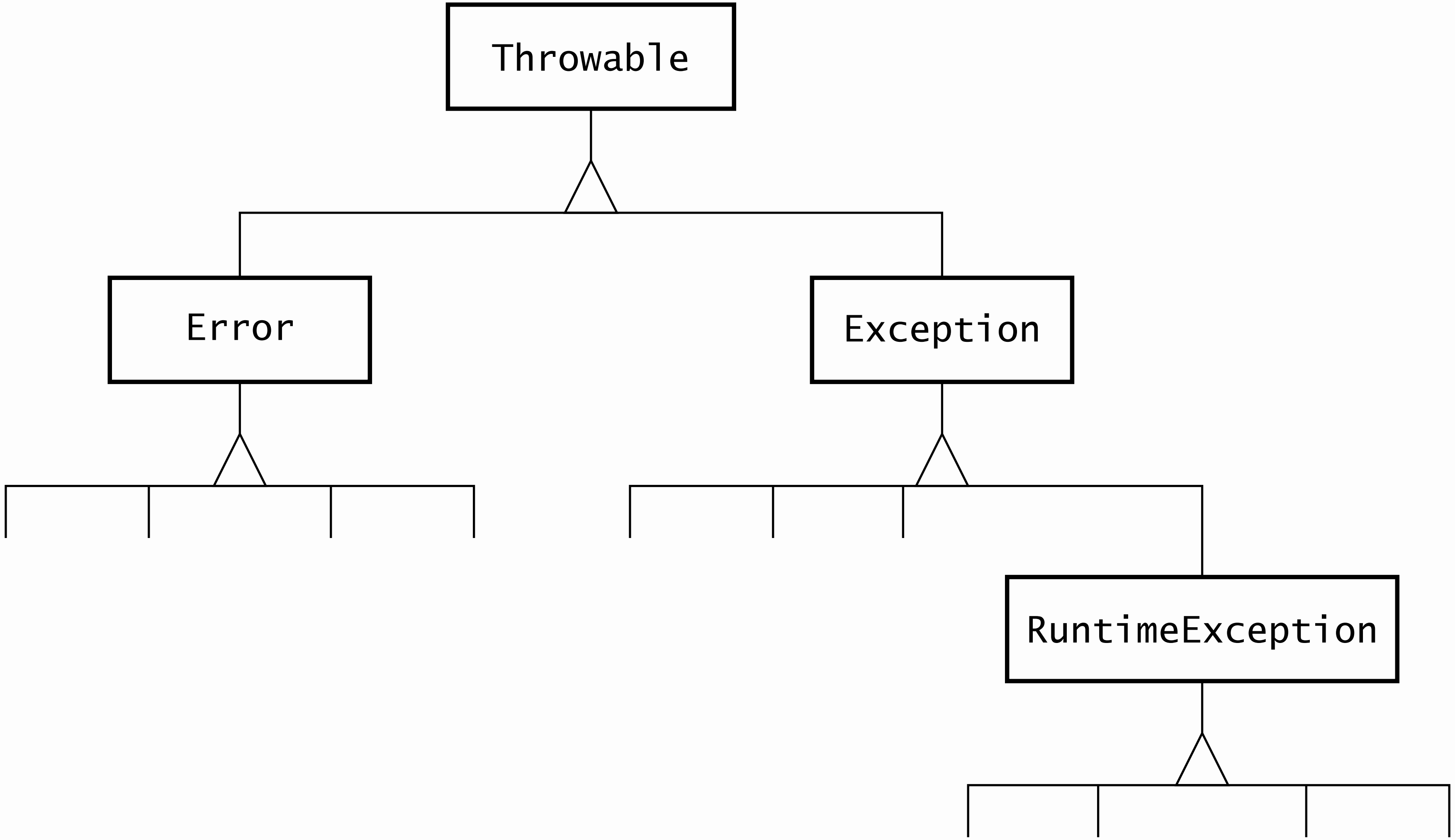
> **Note:** When a method is overridden in a subclass, it must not throw exceptions not specified in the superclass.

There are two types of exceptions:

- **runtime exceptions** – exceptions that can occur almost everywhere, they are usually produced directly by the runtime system (arithmetic exceptions, pointer exceptions, indexing exceptions).
- **checked exceptions** – all other exceptions (including user defined exceptions).

The compiler checks that **checked exceptions** are either caught or specified. **Runtime exceptions** need not be caught or specified.

Hierarchy of Exceptions



Hierarchy of Exceptions (cont.)

Subclasses of **Throwable**:

- Subclasses of **Error** – exceptions of that indicates serious problems that a reasonable application should not try to catch.
- Subclasses of **Exception** – “normal” exceptions that a reasonable application might want to catch. User-defined exceptions should be subclasses of **Exception** (but not of **RuntimeException**).
- Subclasses of **RuntimeException** – runtime exceptions, usually produced by the runtime system. An application might want to catch them.

Note: The classes **Throwable**, **Error**, **Exceptions**, and **RuntimeException** are from the package **java.lang**.

It is not necessary to catch or specify subclasses of **Error** and **RuntimeException**. All other exceptions must be either caught or specified.

Hierarchy of Exceptions (cont.)

It is convenient to hierarchize exceptions using inheritance. This approach enables:

- grouping of error types
- error differentiation.

```
public class StackException extends Exception {
    public StackException(String message) {
        super(message);
    }
}
```

```
public class EmptyStackException extends StackException {
    public EmptyStackException() {
        super("The stack is empty.");
    }
}
```

Throwing an Exception

Any Java code can throw an exception using the **throw** statement:

```
throw someThrowableObject;
```

The **throw** statement requires a single argument – a throwable object.

An example of throwing an exception in an implementation of a stack:

```
public Object pop() throws EmptyStackException {
    if (size == 0) {
        throw new EmptyStackException();
    }
    Object obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

Class Throwable

The constructors of **java.lang.Throwable**:

- Throwable()
- Throwable(String message)
- Throwable(String message, Throwable cause)
- Throwable(Throwable cause)

The most important methods:

- String getMessage()
- Throwable getCause()
- Throwable initCause(Throwable cause)
- String toString()
- void printStackTrace()

Note: Every exception contains information about the call stack at the moment when the exception was created.

Class Error

An **Error** is a subclass of **Throwable** that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

The most important subclasses (in the package **java.lang**):

- VirtualMachineError
  - OutOfMemoryError
  - StackOverflowError
  - InternalError
  - UnknownError
- LinkageError
- ThreadDeath
- AssertionError

Class RuntimeException

The most important subclasses of **java.lang.RuntimeException**:

- ArithmeticException
- IndexOutOfBoundsException
  - ArrayIndexOutOfBoundsException
  - StringIndexOutOfBoundsException
- IllegalArgumentException
  - NumberFormatException
- NullPointerException
- ClassCastException
- NegativeArraySizeException
- ArrayStoreException
- IllegalStateException
- UnsupportedOperationException

Exception Advantages

The use of exceptions has the following advantages over traditional error management techniques:

- **Separating error handling code from “regular” code**
  A problem which can raise at many places in program can be handled in only one place.
- **Propagating errors up the call stack**
  Mechanism enabling propagation of exceptions over the call stack enables transparent handling of errors raised in libraries.
- **Grouping error types and error differentiation**
  Multiple types of errors can be handled similarly at one place.

Streams

Often a program needs to:

- bring in information from an external source, or
- send out information to an external destination.

The information can be:

- in a file on a disk
- somewhere on the network
- in memory
- in another program

Streams present an abstraction that allows to access (read or write) such information **sequentially.**

Using the streams we can access sources and destinations of information in a unified way no matter where they actually are.

Streams

We distinguish two types of streams:

- **input streams** – programs read from them
- **output streams** – programs write to them

The algorithms for sequentially reading and writing data are basically the same:

- **Reading**

```
open a stream
while more information
    read information
close the stream
```

- **Writing**

```
open a stream
while more information
    write information
close the stream
```

Streams

The package **java.io** contains a collection of stream classes.

The stream classes are divided into two class hierarchies:

- **Byte Streams** – they work on streams of 8-bit bytes (binary data). They are subclasses of (abstract) classes:
  - **InputStream** – input streams
  - **OutputStream** – output streams

- **Character Streams** – they work on streams of 16-bit characters (text files). They are subclasses of (abstract) classes:
  - **Reader** – input streams
  - **Writer** – output streams

Streams

InputStream:

- int read()
- int read(byte[] b)
- int read(byte[] b, int off, int len)

OutputStream:

- void write(int b)
- void write(byte[] b)
- void write(byte[] b, int off, int len)

Reader:

- int read()
- int read(char[] cbuf)
- int read(char[] cbuf, int off, int len)

Writer:

- void write(int c)
- void write(char[] cbuf)
- void write(char[] cbuf, int off, int len)

Streams

There are also other methods. All these classes contain method

- void close()

  Note: The method `close()` can be called either explicitly, or implicitly by the garbage collector.

The classes **InputStream** and **Reader** contain methods

- long skip(long n)
- boolean markSupported()
- void mark(int readAheadLimit)
- void reset()

The classes **OutputStream** and **Writer** contain method

- void flush()

Most of the methods that work with streams can throw **java.io.IOException** (or some of its subclasses).

File Streams

The file streams read or write a file on the file system:

- FileInputStream
- FileOutputStream
- FileReader
- FileWriter

An example of use of **FileReader** and **FileWriter**:

```
Reader in = new FileReader("input.txt");
Writer out = new FileWriter("output.txt");
int c;
while ((c = in.read()) >= 0) {
    out.write(c);
}
in.close();
out.close();
```

File Streams

It is better to read and write bigger chunks of data:

```
InputStream in = new FileInputStream("input.txt");
OutputStream out = new FileOutputStream("output.txt");
final int BUF_LEN = 8192;
byte[] buf = new byte[BUF_LEN];
int l;
while ((l = in.read(buf, 0, BUF_LEN)) >= 0) {
    out.write(buf, 0, l);
}
in.close();
out.close();
```

File Streams

File streams can be created using:

- a file name (class **String**)
- a file object (class **File**)
- a file descriptor (class **FileDescriptor**)

For example, the class **FileReader** contains the following constructors:

- FileReader(String fileName)
- FileReader(File file)
- FileReader(FileDescriptor fd)

Classes **FileOutputStream** and **FileWriter** contain also constructors that allow to specify if an existing file should be overwritten or data should be appended to it:

- FileOutputStream(String name, boolean append)
- FileOutputStream(File file, boolean append)

Class File

The instances of the class **java.io.File** represent files on the file system.

It presents an abstract, system-independent view of hierarchical pathnames.

We can create a `File` object for a file on the file system and query the object for information about the file, such as:

- the full path name
- the name of its parent directory
- if it is directory or a regular file
- if it is an absolute or relative pathname
- if the file exists
- the length of the file
- the access rights (if it can be read and/or written)
- other attributes (time of modification, if it is hidden, ...)

Class `File` (cont.)

We can use an object of class `File` also for manipulation with the given file. We can for example:

- create the file
- delete the file
- rename the file
- obtain a list of files in the directory
- create a subdirectory
- set time of modification
- create temporary files

Example of a deletion of a file:

```
String filename = "test.txt";
File f = new File(filename);
boolean ok = f.delete();
System.out.println(ok ? "O.K." : "Not deleted");
```

Class File (cont.)

Example of use of the class `File`:

```
File input = new File("input.txt");
if (!input.exists()) {
   System.err.println("Error: file \"" + input.getName() +
      "\" doesn't exist");
   return;
}

FileReader reader = new FileReader(input);
...
```

Filter Streams

The **java.io** package provides a set of abstract classes that define and partially implement **filter streams**:

- FilterInputStream
- FilterOutputStream
- FilterReader
- FilterWriter

Filter streams allow to combine features of streams and achieve desired functionality.

A filter stream is constructed on another stream (the **underlying** stream):

- The `read` method reads input from the underlying stream, filters it and passes to the caller.
- The `write` method filters output and writes the resulting data to the underlying stream.

Buffered Streams

An example of filter streams are **buffered streams**:

- BufferedInputStream
- BufferedOutputStream
- BufferedReader
  - LineNumberReader
- BufferedWriter

An example of use of BufferedReader:

```
BufferedReader reader =
   new BufferedReader(new FileReader("input.txt"));
String s;
while((s = reader.readLine()) != null) {
   System.out.println(s);
}
```

Other Types of Streams

Another type of filter streams are **pushback streams**:

- PushbackInputStream
- PushbackReader

They add to streams the ability to “push back” or “unread” bytes or characters.

The are streams for conversion between byte streams and character streams:

- InputStreamReader
- OutputStreamWriter

  Note: The character encoding used by these streams can be specified in their constructors.

```
Reader r = new InputStreamReader(
           new FileInputStream("input.txt"));
Writer w = new OutputStreamWriter(
           new FileOutputStream("output.txt"), "iso-8859-2");
```

Print Streams

Print streams allow to print values of different data types in a human readable form:

- PrintStream
- PrintWriter

Unlike other streams the print streams never throw an `IOException`; instead, exceptional situations merely set an internal flag that can be tested via the `checkError()` method.

Optionally, they can be created so as to flush automatically after every end of line.

The overloaded methods `print()` and `println()` are used to print values of various data types:

- void print(boolean b)
- void print(char c)
- void print(int i)
- . . .

Print Streams (cont.)

The methods `println()` should be used to print line separators instead of using `'\n'` in printed strings.

In the following example

```
PrintWriter w = new PrintWriter(
                new FileOutputStream("output.txt"));
w.print("Hello\n");
```

it is better to use

```
w.println("Hello");
```

Different platforms use different line separators:

Platform	Decimal	Chars
MS Windows	13 10	"\r\n"
Unix	10	"\n"
MacOS	13	"\r"

Standard Input and Output

Three standard streams are streams are defined in the class **java.lang.System** as static final variables:

- **in** – standard input (`InputStream`)
- **out** – standard output (`PrintStream`)
- **err** – standard error output (`PrintStream`)

All these streams are implicitly opened.

These streams should not be closed.

Standard input stream typically corresponds to keyboard input.

Standard output and error streams typically correspond to display output.

All these streams can be redirected by a user to a file or another program:

```
$ java MyClass < input.txt > output.txt
$ java MyClass < input.txt | less
```

Stream Tokenizer

The StreamTokenizer class takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time. The stream tokenizer can recognize identifiers, numbers, quoted strings, and various comment styles.

```
StreamTokenizer s = new StreamTokenizer(
        new InputStreamReader(System.in));
s.eolIsSignificant(true);
loop: while (true) {
    switch (s.nextToken()) {
    case StreamTokenizer.TT_EOF: break loop;
    case StreamTokenizer.TT_WORD:
      System.out.println("a word: " + s.sval); break;
    case StreamTokenizer.TT_NUMBER:
      System.out.println("a number: " + s.nval); break;
    case StreamTokenizer.TT_EOL:
      System.out.println("EOL"); break;
    default:
      System.out.println("other: " + (char)s.ttype);
    }
}
```

Reading from URL

The streams are also used to represent network connections:

```
URL url = new URL("http://java.sun.com/docs");
InputStream in = url.openStream();
OutputStream out = new FileOutputStream("output.txt");
int c;
while ((c = in.read()) >= 0) {
    out.write(c);
}
in.close();
out.close();
```

Note: The class **URL** is from the **java.net** package.

Data Streams

There are input and output streams for reading and writing primitive data types in a **binary** (but portable) format:

- DataInputStream
- DataOutputStream

The class **DataInputStream** contains methods such as:

- void readFully(byte[] b)
- void readFully(byte[] b, int off, int len)
- boolean readBoolean()
- byte readByte()
- int readUnsignedByte()
- short readShort()
- int readUnsignedShort()
- int readInt()
- String readUTF()
- . . .

Data Streams (cont.)

The class **DataOutputStream** contains methods such as:

- void writeBoolean(boolean v)
- void writeByte(int v)
- void writeChar(int v)
- void writeInt(int v)
- void writeLong(long v)
- void writeFloat(float v)
- void writeDouble(double v)
- void writeBytes(String s)
- void writeChars(String s)
- void writeUTF(String str)

All these methods for reading and writing binary data are declared in interfaces:

- DataInput
- DataOutput

Serialization

Java's **object serialization** allows to take any object that implements the **java.io.Serializable** interface and turn it into a sequence of bytes that can later be fully restored to regenerate the original object.

The following classes are used to read and write objects:

- ObjectInputStream
- ObjectOutputStream

It is possible to use these classes to read and write primitive data types since they implement interfaces `DataInput` and `DataOutput`.

> **Note:** Instance variables defined as **`transient`** and static variables are prevented from serialization.

The interface **java.io.Serializable** does not declare any methods.

Serialization (cont.)

Writing into an object stream:

```java
FileOutputStream fos = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(new Date());
oos.close();
```

Reading from an object stream:

```java
FileInputStream fis = new FileInputStream("t.tmp");
ObjectInputStream ois = new ObjectInputStream(fis);
int i = ois.readInt();
String today = (String) ois.readObject();
Date date = (Date) ois.readObject();
ois.close();
```

Serialization (cont.)

Classes that require special handling during the serialization and deserialization process must implement two special methods with the given signatures:

```
private void writeObject(ObjectOutputStream s)
        throws IOException {
    s.defaultWriteObject();
    // customized serialization code
}
```

```
private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    // customized deserialization code
    //   . . .
    // followed by code to update the object, if necessary
}
```

Serialization (cont.)

For complete, explicit control of the serialization process, a class must implement the **java.io.Externalizable** interface.

For Externalizable objects, only the identity of the object's class is automatically saved by the stream. The class is responsible for writing and reading its contents.

```
package java.io;

public interface Externalizable extends Serializable {
    public void writeExternal(ObjectOutput out)
        throws IOException;

    public void readExternal(ObjectInput in)
        throws IOException, java.lang.ClassNotFoundException;
}
```

Note: Default constructor of a deserialized object implementing Externalizable is always invoked. Thus the constructor must be **public**.

Random Access Files

The input and output streams are **sequential access streams**. **Random access files** permit nonsequential, or random, access to a file's contents.

The **RandomAccessFile** class in the **java.io** package implements a random access file.

> **Note:** The **RandomAccessFile** class is not part of class hierarchy of streams, but it implements **DataInput** and **DataOutput** interfaces.

It is possible to open a random access file only for reading:

```
new RandomAccessFile("file.txt", "r");
```

And also for reading and writing:

```
new RandomAccessFile("file.txt", "rw");
```

After the file has been opened, the common methods `read()` and `write()` can be used for reading and writing.

Random Access Files (cont.)

The class `RandomAccessFile` supports the notion of a **file pointer** that indicates the current location in the file.

- When the file is opened, the file pointer is set to 0 (to the beginning of the file).
- Calls to the `read()` and `write()` methods adjust the file pointer by the number of bytes read or written.

The `RandomAccessFile` class contains three methods for explicitly manipulating the file pointer:

- int skipBytes(int n) – moves the file pointer forward the specified number of bytes
- void seek(long pos) – positions the file pointer just before the specified byte
- long getFilePointer() – returns the current byte location of the file pointer

Random Access Files (cont.)

The `RandomAccessFile` class contains also methods for manipulation with the length of the file:

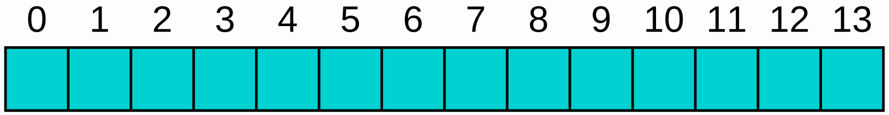
- long length() – returns the length of the file
- void setLength(long newLength) – sets the length of the file

Data Structures

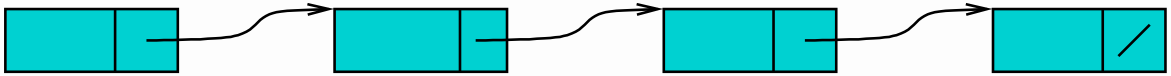
The basic data structures are:

- array
- list
- hashtable
- tree

Array: indexed access, can be resizable

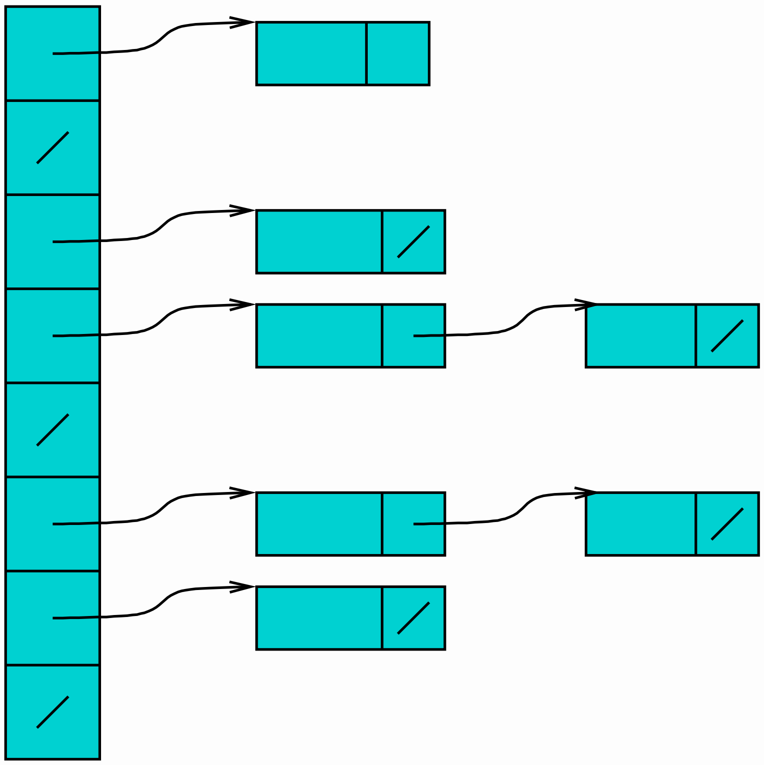


List: singly or doubly linked, can be circular

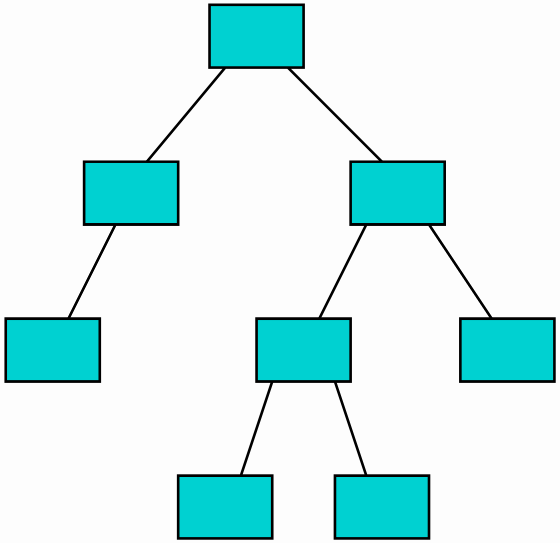


Data Structures (cont.)

Hashtable:



Tree:



Abstract Data Types

Data structures support different operations:

- insert an element
- remove an element
- search an element
- . . .

Abstract data types are interfaces specifying what operations are provided. Examples of ADTs:

- **Set**
- **Dictionary** – also called **Map**
- **Vector** – resizable array
- **Stack** – also called **LIFO**
- **Queue** – also called **FIFO**
- **Priority Queue**

Collections in Java

A **collection** (sometimes called **container**) is an objects that groups multiple elements into single unit.

Earlier versions of Java included the following collections:

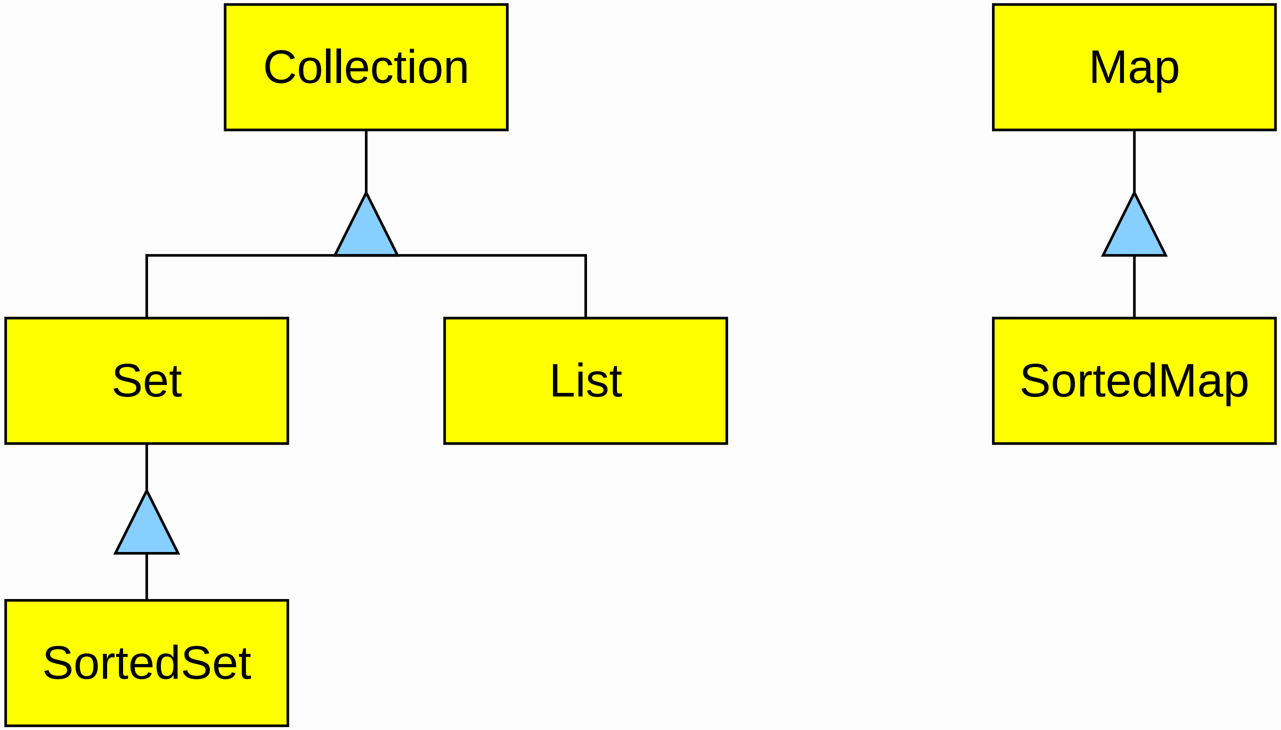
- java.util.Vector
- java.util.Hashtable
- array

Current versions of Java contain **collection framework** – a unified architecture for representing and manipulating collections. It consists of:

- **Interfaces** – abstract data types representing collections
- **Implementations** – concrete implementations of the interfaces
- **Algorithms** – methods that perform useful computations (searching and sorting)

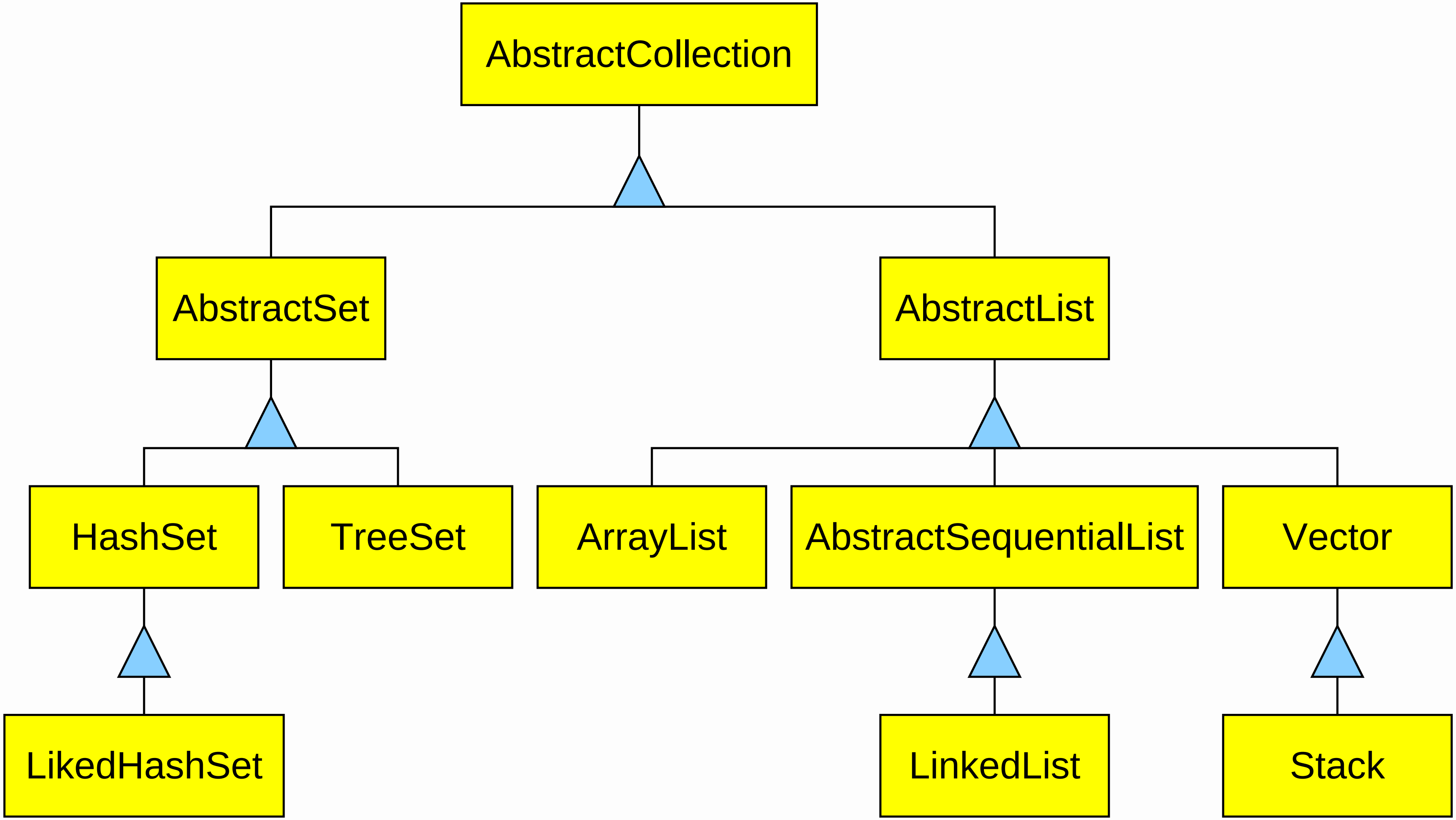
Interfaces

The collection interfaces in the package **java.util** form a hierarchy:



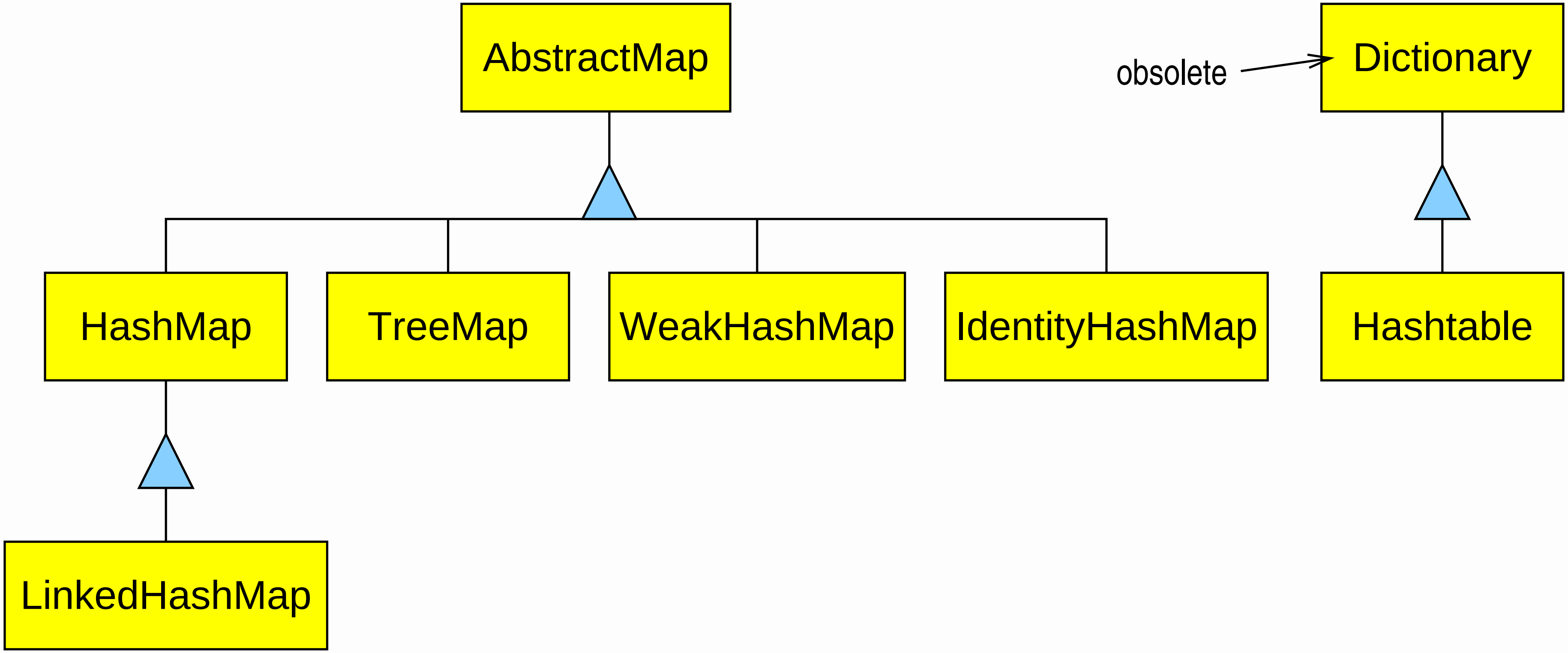
Implementations

The classes implementing collections in the package **java.util**:



Implementations (cont.)

The classes implementing maps:



The Collection Interface

The **Collection** is the root of the collection hierarchy.

A **Collection** represents a group of objects – its **elements**. (Some implementations allow duplicate elements and others do not.)

The primary use of the **Collection** interface is pass around collections of objects where maximum generality is desired.

The **Collection** interface declares the following basic operations:

- int size()
- boolean isEmpty()
- boolean contains(Object o)
- boolean add(Object o) – *optional*
- boolean remove(Object o) – *optional*
- Iterator iterator()

Note: Some operation are designated as **optional**. Implementations that do not implement them throw an **UnsupportedOperationException**.

Iterators

An **iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The **java.util.Iterator** provides uniform interface for traversing different aggregate structures.

```
public interface Iterator {
   boolean hasNext();
   Object next();
   void remove();   // optional
}
```

Example of use:

```
Collection c = new ArrayList();
   . . .    // fill the collection
for (Iterator i = c.iterator(); i.hasNext(); ) {
    Object o = i.next();
       . . .  // process the element
}
```

Enumerations

Earlier implementations of Java used the **java.util.Enumeration** interface instead of **iterator**:

```
public interface Enumeration {
    boolean hasMoreElements();
    Object nextElement();
}
```

The differences between them are:

- **Iterator** allows the caller to remove elements from the underlying collection.
- Method names have been improved in **Iterator**.

New implementations should use **Iterator** in preference to **Enumeration**.

Iterators (cont.)

The **Iterator** interface contains the optional method `remove()` that removes from the underlying collection the last element that was returned by `next()`:

- The `remove()` method may be called only once per call to `next()` – an exception is thrown if this condition is violated.
- The `remove()` method is the **only** safe way to modify a collection during iteration.
- The behavior is unspecified if the underlying collection is modified in any other way while iteration is in progress.

Bulk Operations

The **bulk operations** perform some operation on an entire **Collection** in a single shot:

- boolean containsAll(Collection c)
- boolean addAll(Collection c) – *optional*
- boolean removeAll(Collection c) – *optional*
- boolean retainAll(Collection c) – *optional*
- void clear() – *optional*

For example. to remove **all** instances of a specified element e from a collection c we can use:

```
c.removeAll(Collections.singleton(e));
```

Note: The class **Collections** contains many useful static methods that operate on collections. The `singleton()` method returns an immutable collection (set) containing only the specified object.

Array Operations

The `toArray()` allow the contents of a **Collection** to be translated into an array:

- Object[] toArray()
- Object[] toArray(Object[] a)

The following code dumps the contents of c into a newly allocated array:

```
Object[] a = c.toArray();
```

Suppose c is a collection known to contain only strings. The following code can be used to dump the contents of c into a newly allocated array of `String`:

```
String[] a = (String[])c.toArray(new String[0]);
```

Note: If the collection fits in the specified array, this array is used, otherwise a new array is allocated.

The Set Interface

A **Set** is a **Collection** that cannot contain duplicate elements. It models a mathematical **set** abstraction.

The **Set** interface contains **no** methods than those inherited from **Collection**.

There are two general-purpose **Set** implementations:

- **HashSet** – stores its elements in a hashtable, it is the best-performing implementation.
- **TreeSet** – stores its elements in a red-black tree, guarantees the order of iteration (the elements will be sorted).

The following code creates a new collection containing the same elements as the collection c, but with all duplicates eliminated:

```
Collection d = new HashSet(c);
```

The Set Interface (cont.)

Example of use of a **Set** that prints out any duplicate words, the number of distinct words, and a list of the words with duplicates eliminated:

```java
import java.util.*;

public class FindDuplicates {
    public static void main(String[] args) {
        Set s = new HashSet();
        for (int i = 0; i < args.length; i++) {
            if (!s.add(args[i])) {
                System.out.println("Duplicate detected: "
                                   + args[i]);
            }
        }
        System.out.println(s.size() +
                           " distinct words detected: " + s);
    }
}
```

The Set Interface (cont.)

The bulk operations on sets correspond to standard set-algebraic operations:

- `s1.containsAll(s2)` – returns **true** if s2 is a **subset** of s1

  $$s_2 \subseteq s_1$$

- `s1.addAll(s2)` – transforms s1 into the **union** of s1 and s2

  $$s_1 \cup s_2$$

- `s1.retainAll(s2)` – transforms s1 into the **intersection** of s1 and s2

  $$s_1 \cap s_2$$

- `s1.removeAll(s2)` – transforms s1 into the **set difference** of s1 and s2

  $$s_1 - s_2$$

The List Interface

A **List** is an ordered **Collection** (sometimes called a **sequence**). Lists may contain duplicate elements.

There are two general-purpose **List** implementations:

- **ArrayList** – generally the best-performing implementation
- **LinkedList** – offers better performance under certain circumstances

The **List** contains methods for positional access that manipulate elements based on their numerical position in the list:

- Object get(int index)
- Object set(int index, Object element) – *optional*
- void add(int index, Object element) – *optional*
- Object remove(int index) – *optional*
- boolean addAll(int index, Collection c) – *optional*

The List Interface (cont.)

For example, the following method swaps two elements of a list:

```
private static void swap(List a, int i, int j) {
    Object tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

The following method randomly permutes the specified **List** using the specified source of randomness:

```
public static void shuffle(List a, Random rnd) {
    for (int i = a.size(); i > 1; i--) {
        swap(a, i-1, rnd.nextInt(i));
    }
}
```

Note: The class **Collections** contains such method shuffle().

The List Interface (cont.)

- The remove() operation always removes the **first** occurrence of the specified element.
- The add() and addAll() operations always append the new element(s) to the **end** of the list.
- To concatenate one list to another we can use:

```
list1.addAll(list2);
```

- The non-destructive version of concatenation:

```
List list3 = new ArrayList(list1);
list3.addAll(list2);
```

- The **List** interface contains two methods for searching:
  - int indexOf(Object o)
  - int lastIndexOf(Object o)

The ListIterator Interface

The **List** interface supports its own extended version of iterator:

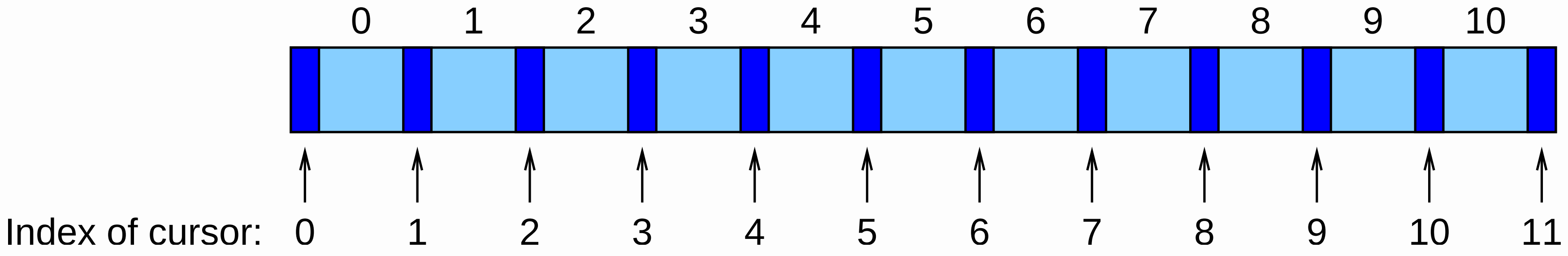
```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

    void remove();          // optional
    void set(Object o);     // optional
    void add(Object o);     // optional
}
```

The `ListIterator` Interface (cont.)

To obtain `ListIterator` we can use **List** methods:

- ListIterator listIterator()
- ListIterator listIterator(int index)

A list iterators has a cursor pointing **between** elements:



The ListIterator Interface (cont.)

Iterating backwards in a list:

```
for (ListIterator i = list.listIterator(list.size());
        i.hasPrevious(); ) {
    Object o = i.previous();
     . . .
}
```

A method that replaces all occurrences of one specified value with another:

```
public static void replace(List l, Object x, Object y) {
    for (ListIterator i = l.listIterator(); i.hasNext(); ) {
        if (x == null ? i.next() == null
                      : x.equals(i.next())) {
            i.set(y);
        }
    }
}
```

The List Interface (cont.)

The **List** interface contains a method returning a **range-view**:

- List subList(int fromIndex, int toIndex)

The returned **List** contains the portion of the original list whose indexes range from fromIndex, inclusive, to toIndex, exclusive.

Changes in the former **List** are reflected in the latter.

For example, to remove a range of elements from a list we can use:

```
list.subList(fromIndex, toIndex).clear();
```

Searching for an element in a range:

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

The Collections Class

The **Collections** class contains static methods implementing different algorithms working on collections. Most of them apply specifically to **List**:

- void sort(List list)
- int binarySearch(List list, Object key)
- void reverse(List list)
- void shuffle(List list)
- void fill(List list, Object obj)
- void copy(List dest, List src)

There is a similar class called **Arrays** containing as static methods algorithms working on arrays.

The Map Interface

A **Map** is an object that maps keys to values.

A map cannot contain duplicate keys: Each key can map to at most one value.

The most important methods:

- Object put(Object key, Object value) – *optional*
- Object get(Object key)
- Object remove(Object key) – *optional*
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- int size()
- boolean isEmpty()

The Map Interface (cont.)

Other methods:

- void putAll(Map t) – *optional*
- void clear() – *optional*
- Set keySet()
- Collection values()
- Set entrySet()

The **Collection**–view methods provide the **only** means to iterate over a **Map**:

```
for (Iterator i = m.keySet().iterator(); i.hasNext(); ) {
    System.out.println(i.next());
}
```

The Map Interface (cont.)

There are two general-purpose **Map** implementations:

- **HashMap** - stores its entries in a hash table, it is the best-performing implementation
- **TreeMap** - stores its entries in a red-black tree, guarantees the order of iteration

There is also an older class **Hashtable.**

Hashtable has been retrofitted to implement **Map.**

Object Ordering

Objects that implement the **java.lang.Comparable** interface can be ordered automatically. The **Comparable** interface provides **natural ordering** for a class:

```
public interface Comparable {
    public int compareTo(Object o);
}
```

The method o1.compareTo(o2) returns:

- a negative integer – if o1 is less than o2
- zero – if o1 is equal to o2
- a positive integer – if o1 is greater than o2

Many standard classes such as **String** and **Date** implement the **Comparable** interface.

Object Ordering (cont.)

```
import java.util.*;

public class Name implements Comparable {
    private String firstName, lastName;

      . . .

    public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name n = (Name)o;
        return firstName.equals(n.firstName) &&
               lastName.equals(n.lastName);
    }

    public int hashCode() {
        return 31 * firstName.hashCode() +
               lastName.hashCode();
    }

      . . .
```

Object Ordering (cont.)

```
      . . .

    public int compareTo(Object o) {
        Name n = (Name)o;
        int cmp = lastName.compareTo(n.lastName);
        if (cmp != 0) return cmp;
        return firstName.compareTo(n.firstName);
    }
}
```

Note how methods equals() and hashCode() are redefined to be consistent with compareTo().

Comparators

If we want to sort objects in some other order than natural ordering, we can use the **Comparator** interface:

```
public interface Comparator {
    int compare(Object o1, Object o2);
}
```

A **Comparator** is an object that encapsulates ordering.

The compare() method compares two its arguments, returning a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Methods implementing different algorithms in classes **Collections** and **Arrays** allow to specify the comparator that should be used in these algorithms.

The SortedSet Interface

A **SortedSet** is a **Set** that maintains its elements in ascending order, sorted according to the elements' **natural order**, or according to a **Comparator** provided at **SortedSet** creation time.

The **SortedSet** adds the following methods to the methods declared in the **Set** interface:

- SortedSet subSet(Object fromElement, Object toElement)
- SortedSet headSet(Object toElement)
- SortedSet tailSet(Object fromElement)
- Object first()
- Object last()
- Comparator comparator()

The SortedSet Interface (cont.)

There are some differences on behavior of methods inherited from the **Set** interface:

- The iterator returned by the `iterator()` traverses the sorted set in order.
- The array returned by `toArray()` contains the sorted set's elements in order.

The **SortedSet** interface is implemented by the class:

- **TreeSet**

The SortedMap Interface

A **SortedMap** is a **Map** that maintains its entries in ascending order, sorted according to the keys' **natural order**, or according to a **Comparator** provided at **SortedMap** creation time.

The **SortedMap** adds the following methods to the methods declared in the **Map** interface:

- Comparator comparator()
- SortedMap subMap(Object fromKey, Object toKey)
- SortedMap headMap(Object toKey)
- SortedMap tailMap(Object fromKey)
- Object firstKey()
- Object lastKey()

There is one class implementing the **SortedMap** interface:

- **TreeMap**

Implementations

The general-purpose implementations are summarized in the table below:

	Implementations			
	Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

The **SortedSet** and **SortedMap** interfaces are implemented by **TreeSet** and **TreeMap** classes.

The BitSet Class

The **java.util.BitSet** class implements a vector of bits that grows as needed.

Each component of the bit set has a boolean value. The bits of a BitSet are indexed by nonnegative integers. Individual indexed bits can be examined, set, or cleared.

One **BitSet** may be used to modify the contents of another **BitSet** through logical AND, logical inclusive OR, and logical exclusive OR operations.

The **BitSet** class can used as an efficient implementation of a set if the corresponding universe of possible values is finite and small.

The logical operations then correspond to the set operations.

Note: The **BitSet** class is not part of the collection framework.

Nested Classes

It is possible to define a class as a member of another class. Such a class is called **nested class**:

```
class EnclosingClass {
      . . .
    class NestedClass {
          . . .
    }
}
```

A nested class has special privilege: It has unlimited access to its enclosing class's members, even if they are declared private.

A class should be defined within another class when the nested class makes sense only in the context of its enclosing class or when it relies on the enclosing class for its function.

Nested Classes (cont.)

Like other members, a nested class can be declared static.

A static nested class is called just **static nested class**.

A non-static nested class is called an **inner class.**

```
class EnclosingClass {
      . . .
    static class StaticNestedClass {
          . . .
    }

    class InnerClass {
          . . .
    }
}
```

Nested Classes (cont.)

A **static nested class** cannot refer directly to instance variables or methods defined in its enclosing class.

An **inner class** is associated with an instance of its enclosing class and has direct access to that object's instance variables and methods. It cannot define any (non-final) static members itself.

Like other classes, nested classes can be declared **abstract** or **final**.

Also, the access specifiers - **private**, **protected** and **public** - may be used to restrict access to nested classes.

A nested class can be also declared in **any** block of code.

A nested class declared within a method or other smaller block of code has access to any final local variables in scope.

Inner Classes – Example

```
public class Container1 {
   private Object[] items;
     . . .
   public Iterator iterator() {
      return new ContainerIterator();
   }

   class ContainerIterator implements Iterator {
      int index = 0;
      public boolean hasNext() {
         return index < items.length;
      }
      public Object next() {
         if (!hasNext())
            throw new NoSuchElementException();
         return items[index++];
      }
        . . .
   }
}
```

Anonymous Inner Classes

An inner class can be declared without naming it. However, anonymous classes can make code **difficult to read**.

```
public class Container2 {
   private Object[] items;
     . . .
   public Iterator iterator() {
      return new Iterator() {
         int index = 0;
         public boolean hasNext() {
            return index < items.length;
         }
         public Object next() {
            if (!hasNext())
               throw new NoSuchElementException();
            return items[index++];
         }
           . . .
      };
   }
}
```

Anonymous Inner Classes (cont.)

An anonymous class is never **abstract**.

An anonymous class is always an **inner class**, it is never **static**.

An anonymous class is always implicitly **final.**

An anonymous class cannot have an explicitly declared constructor. Instead, the compiler provides an **anonymous constructor**.

Locales

A **java.util.Locale** object represents a specific geographical, political, or cultural region.

An operation that requires a **Locale** to perform its task is called **locale-sensitive** and uses to **Locale** to tailor information for the user.

For example, displaying a number is a locale-sensitive operation - the number should be formatted according to the customs/conventions of the user's native country, region, or culture.

Constructors:

- Locale(String language)
- Locale(String language, String country)
- Locale(String language, String country, String variant)

Some methods:

- static Locale getDefault()
- static Locale[] getAvailableLocales()

Locales (cont.)

Examples:

- **new** `Locale("cs", "CZ")` – Czech, Czech Republic
- **new** `Locale("en", "US")` – English, United States
- **new** `Locale("en", "GB")` – English, United Kingdom
- **new** `Locale("fr", "FR")` – French, France
- **new** `Locale("de", "DE")` – German, Germany

Remark: Locales are often written as "cs_CZ", "en_US", "en_GB", ...

Formatting Numbers

By invoking the methods provided by the **java.text.NumberFormat** we can format numbers, currencies, and percentages according to **Locale**.

Example:

```
NumberFormat f = NumberFormat.getNumberInstance(locale);
String s = f.format(345678.234);
System.out.println(s + " " + locale.toString());
```

We obtain:

```
345 678,234   cs_CZ
345,678.234   en_US
345.678,234   de_DE
```

Similarly we can use methods getCurrencyInstance() and getPercentInstance() to format currencies and percentages.

Formatting Numbers (cont.)

We can use the **java.util.DecimalFormat** class (it is a subclass of **NumberFormat**) to format decimal numbers.

The class allows to control display of leading and trailing zeros, prefixes and suffixes, grouping (thousands) separators, and the decimal separator. The format specified using pattern:

```
String pattern = ...
DecimalFormat f = new DecimalFormat(pattern);
String s = f.format(12345.6789);
System.out.println(pattern + "  " + s);
```

We obtain (using "cs_CZ" locale):

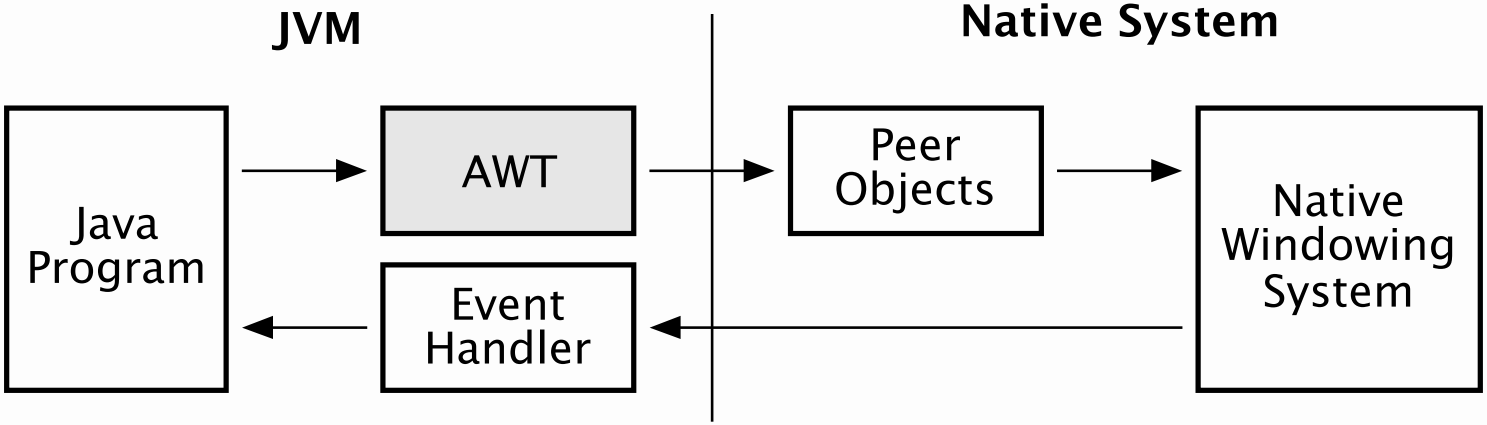
```
###,###.###     12 345,679
###.##          12345,68
000000.000      012345,679
```

It is possible to use the **DecimalFormatSymbols** class to change the symbols that appear in the formatted numbers produced by the `format()` method.

GUI

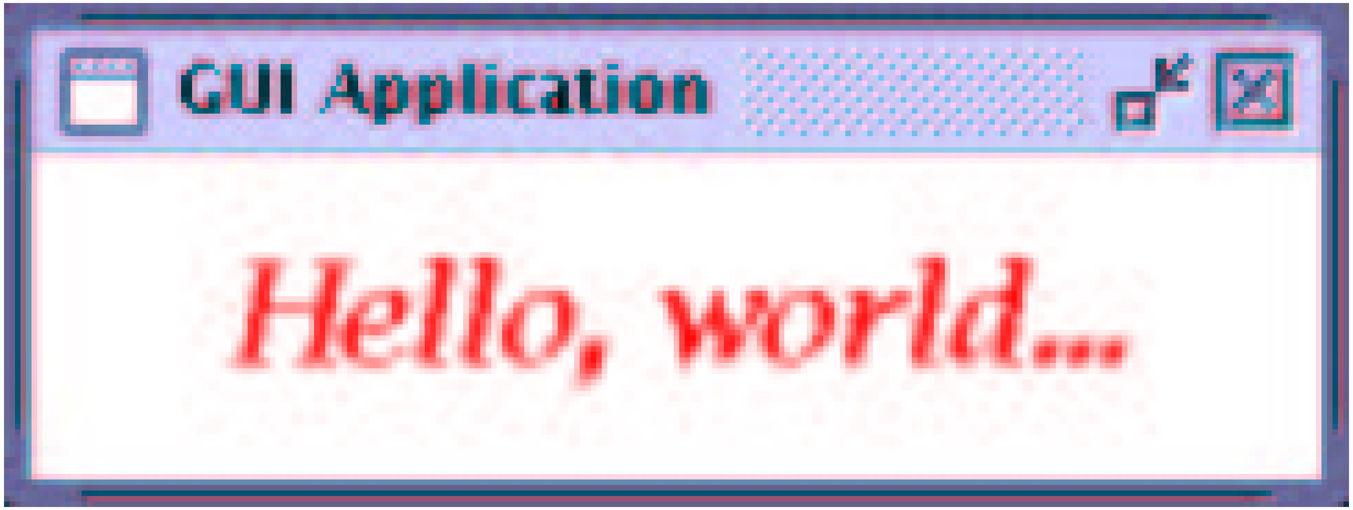
Java supports GUI development through the AWT and the JFC Swing packages.

- **Abstract Window Toolkit (AWT)**
  The AWT provides connection Java application and native GUI. The AWT components depend on native code counterparts (called peers) to handle their functionality.

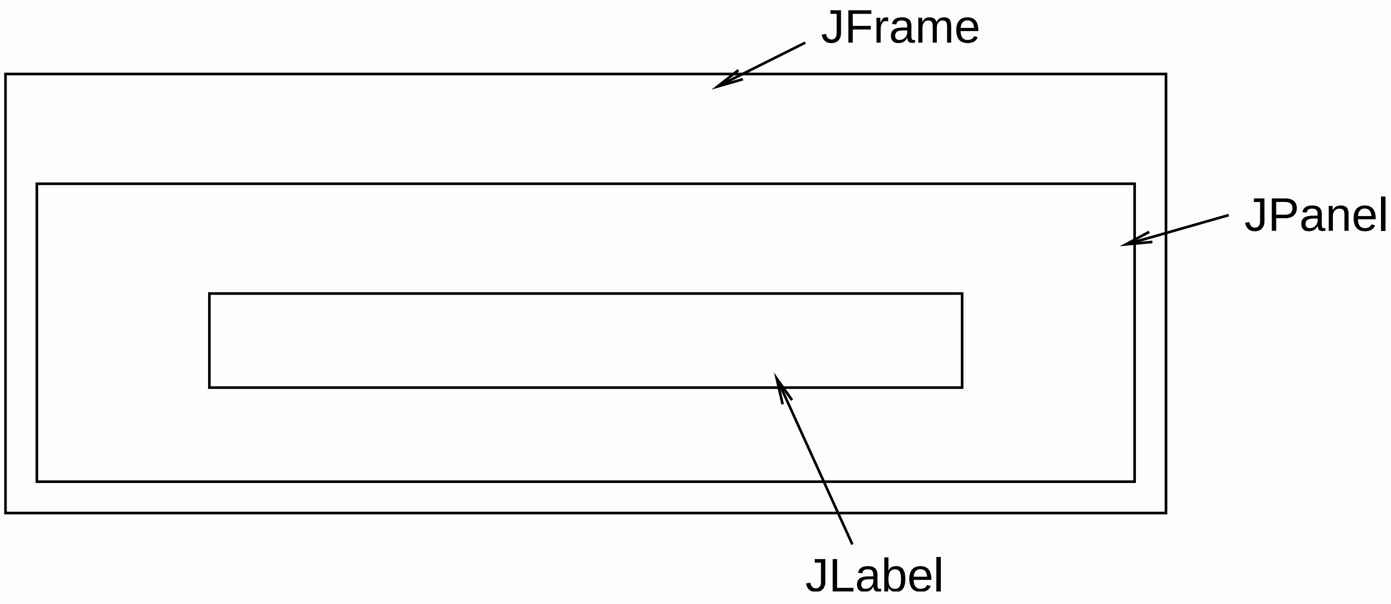


- **JFC Swing**
  Swing implements a set of GUI components that build on AWT technology and provide a pluggable look and feel. Swing is implemented entirely in the Java and do not depend on peers to handle their functionality.

Simple GUI Application



Hello, world...



Simple GUI Application (cont.)

At first, we create a label:

```
JLabel text = new JLabel("Hello, world...");
text.setForeground(Color.RED);
Font  font = new Font("serif", Font.BOLD|Font.ITALIC, 24);
text.setFont(font);
```

Then we create a panel and add the label to the panel:

```
JPanel panel = new JPanel();
panel.setBackground(Color.WHITE);
panel.setBorder(BorderFactory.createEmptyBorder(10, 30,
                10, 30));
panel.add(text);
```

Note: The classes **Color** and **Font** belong to the **java.awt** package, the classes **JLabel** and **JPanel** to the **javax.swing** package.

Simple GUI Application (cont.)

Finally, we create and show a main frame:

```
JFrame.setDefaultLookAndFeelDecorated(true);
JFrame frame = new JFrame("GUI Application");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(panel);
frame.pack();
frame.setVisible(true);
```

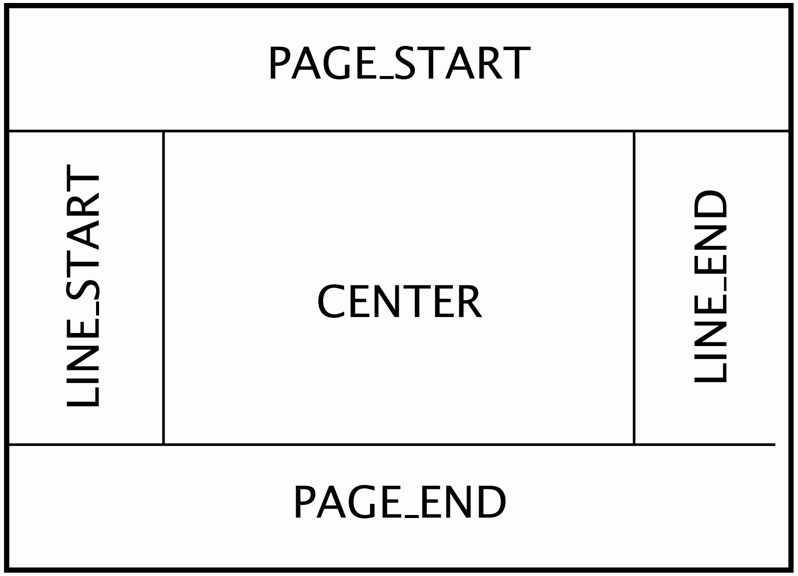
We put all the previous code in a (static) method `createGUI()` and call it from the `main()` method:

```
public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createGUI();
        }
    });
}
```

Layout Management

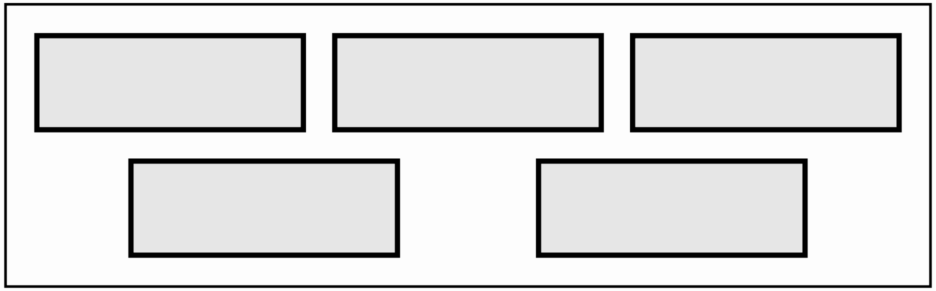
Layout management is a process of determining size and position of components. By default, each container has a layout manager – an object that performs layout management for the components within the container. Java provides several standard layout managers.

- **BorderLayout**
  The `BorderLayout` has five cells called PAGE_START, PAGE_END, `LINE_START`, `LINE_END` and CENTER.

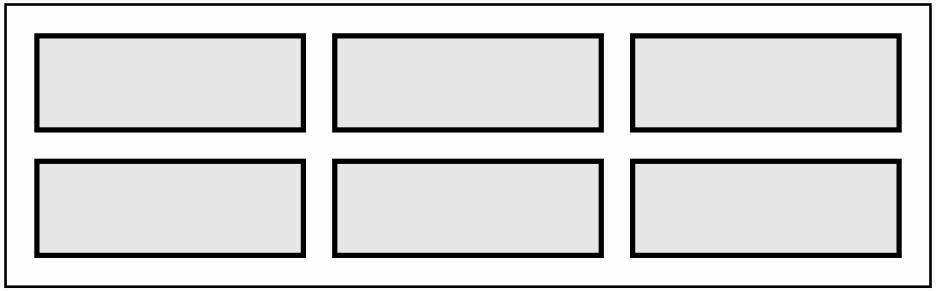


Layout Management (cont.)

- **FlowLayout**
  The simple layout manager "flows" components into the window. The components can be aligned and space between them can be specified.



- **GridLayout**
  The GridLayout manager's strategy is to make each cell exactly the same size so that rows and columns line up in a regular grid.

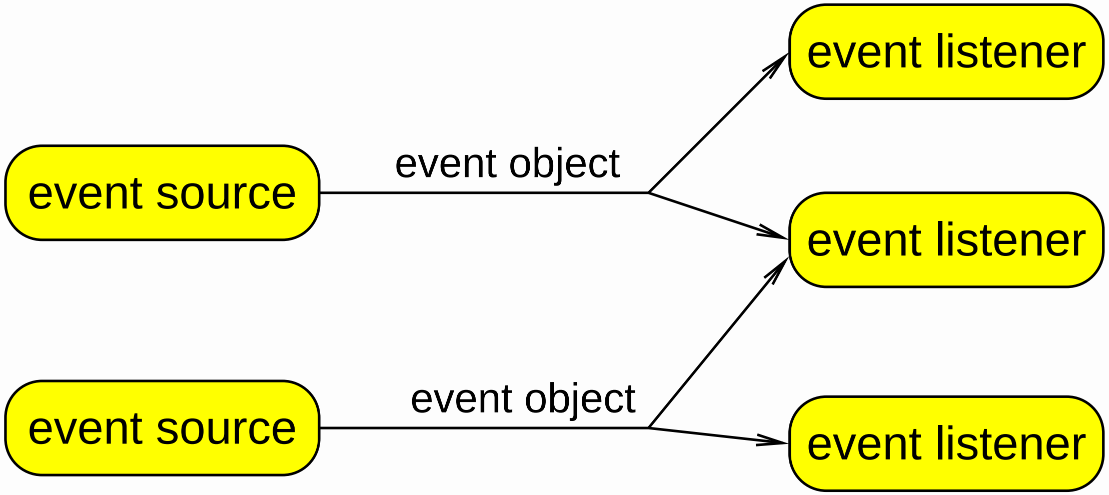


Events and Event Listeners

Evens are instances of subclasses of the **java.util.EventObject** class. They represent informations about particular events that occurred.

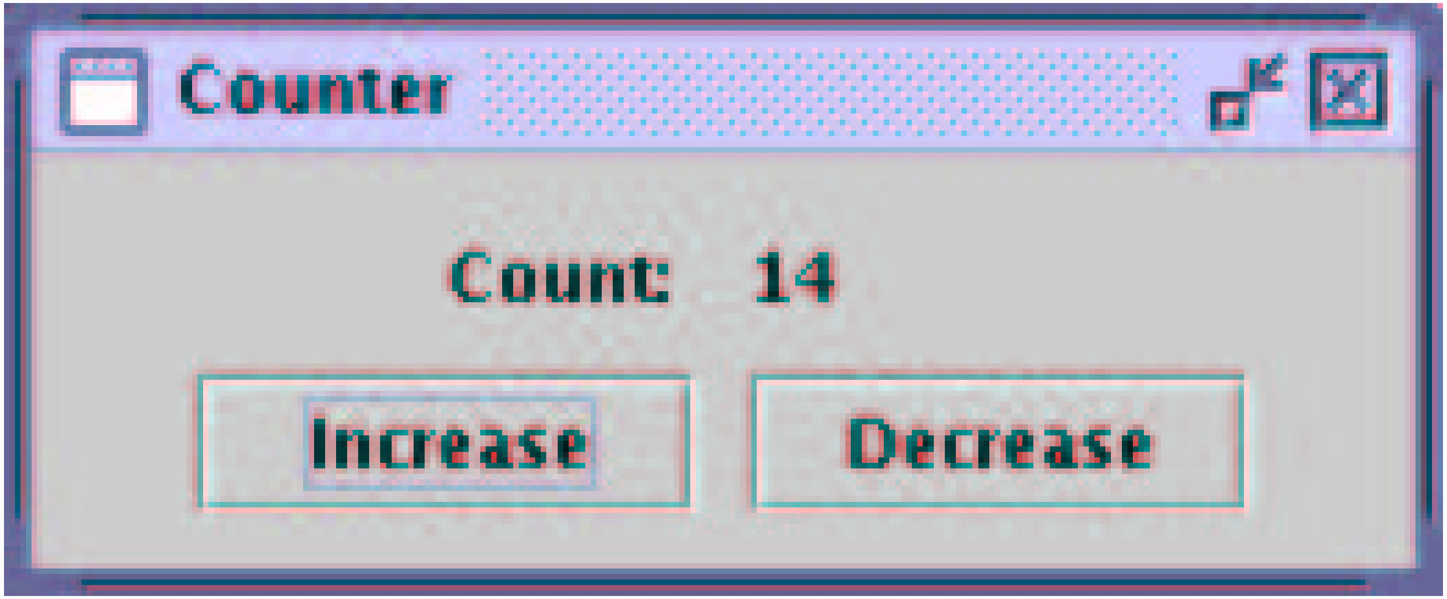
Event Listeners are objects implementing subinterfaces of the **java.util.EventListener** interface.

Most of events and event listener interfaces concerning GUI it defined in the **java.awt.util** package.



Note: Multiple listeners can register to notified of events of a particular type from a particular source. Also the same listener can listen to notifications from different objects.

Example – Counter



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Counter implements ActionListener {
    private int count = 0;
    private JLabel text, value;
    private JButton increase, decrease;
```

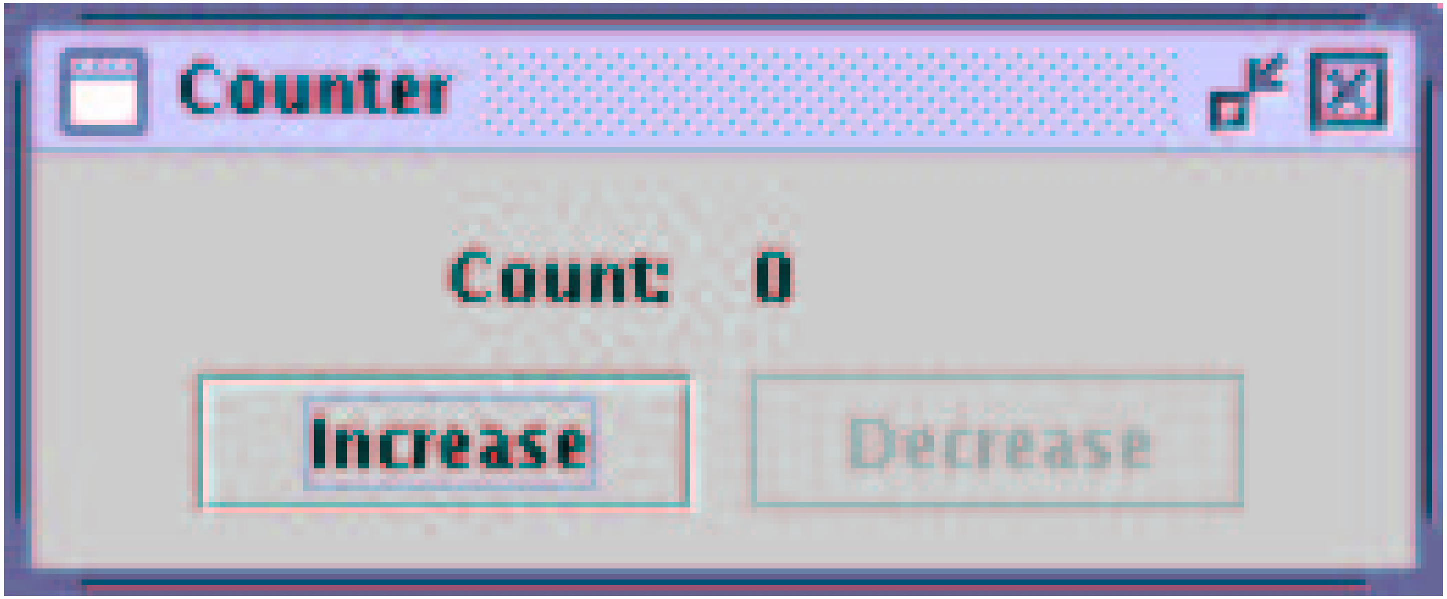
Example – Counter (cont.)

```
public void createGUI() {
    text = new JLabel("Count: ");
    text.setHorizontalAlignment(SwingConstants.RIGHT);
    value = new JLabel("0");
    increase = new JButton("Increase");
    decrease = new JButton("Decrease");
    increase.addActionListener(this);
    decrease.addActionListener(this);
    updateLabel();

    JPanel panel = new JPanel();
    panel.setBorder(BorderFactory.createEmptyBorder(10, 30,
                    10, 30));
    panel.setLayout(new GridLayout(2, 2, 10, 5));
    panel.add(text);
    panel.add(value);
    panel.add(increase);
    panel.add(decrease);
      . . .
```

Example – Counter (cont.)

```
    JFrame.setDefaultLookAndFeelDecorated(true);
    JFrame frame = new JFrame("Counter");
    frame.getContentPane().add(panel);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
}

private void updateLabel() {
    value.setText(Integer.toString(count));
    decrease.setEnabled(count > 0);
}

public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    if (source == increase) {
        count++; updateLabel();
    } else if (source == decrease) {
        if (count > 0) { count--; updateLabel(); }
    }
}
```

Example – Counter (cont.)



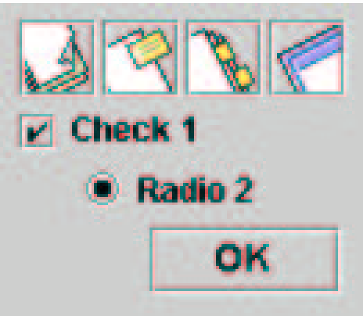
Example – Counter (cont.)

The other possible way how to add action listeners to buttons:

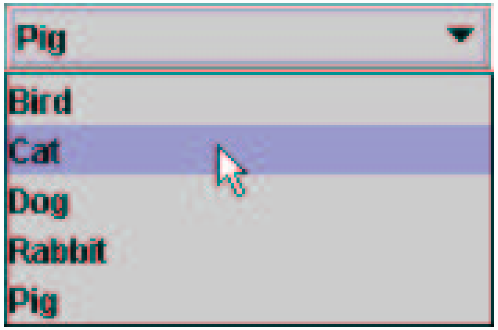
```
increase.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        count++; updateLabel();
    }
});

decrease.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (count > 0) { count--; updateLabel(); }
    }
});
```

Components

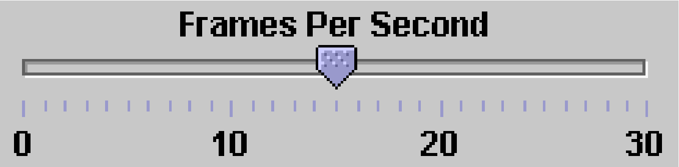
Overview of the most important components from the **javax.swing** package:



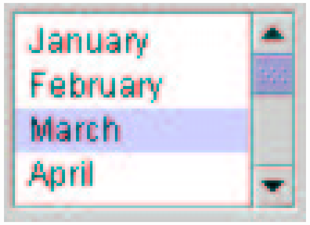
JButton, JCheckBox, JRadioButton



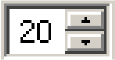
JComboBox



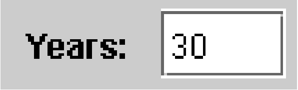
JSlider



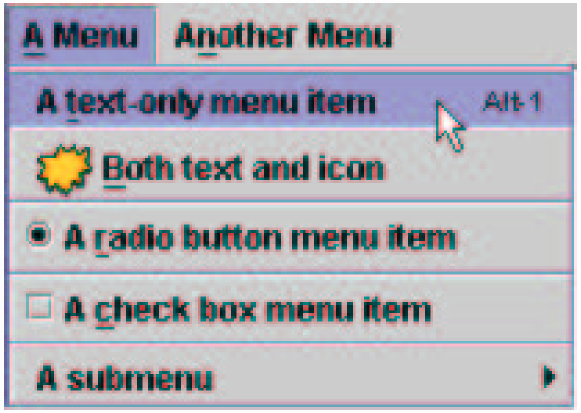
JList



JSpinner

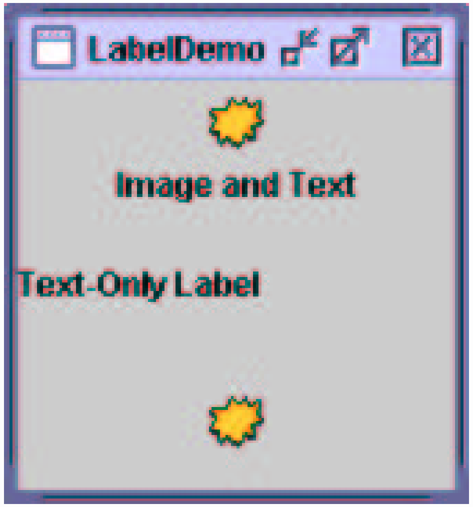


JTextField



JMenu, JMenuItem
JMenuBar

Components (cont.)

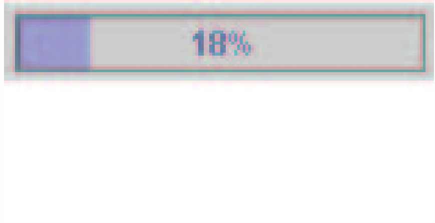


LabelDemo

Image and Text

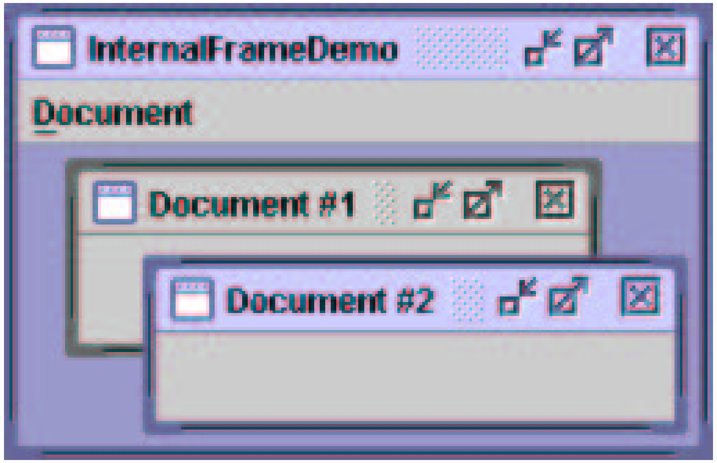
Text-Only Label

JLabel



18%

JProgressBar



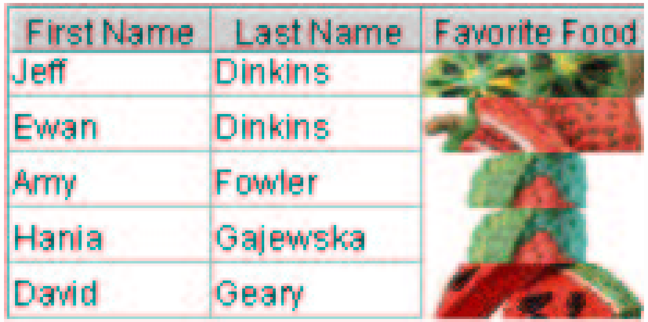
InternalFrameDemo

Document

Document #1

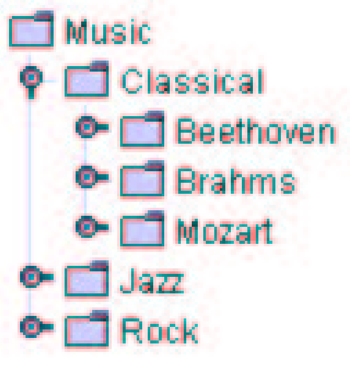
Document #2

JInternalFrame



First Name	Last Name	Favorite Food
Jeff	Dinkins	
Ewan	Dinkins	
Amy	Fowler	
Hania	Gajewska	
David	Geary	

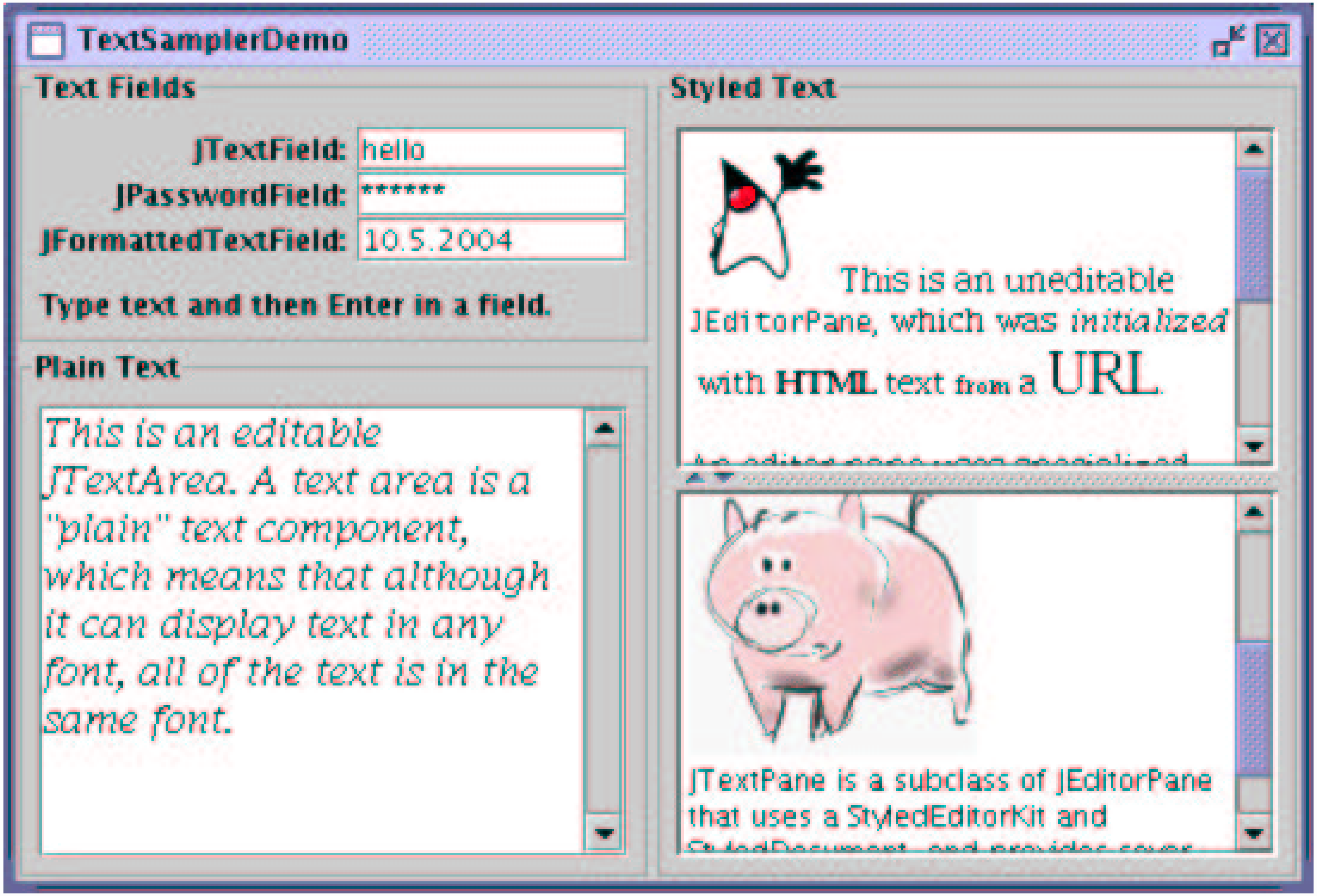
JTable



- Music
  - Classical
    - Beethoven
    - Brahms
    - Mozart
  - Jazz
  - Rock

JTree

Components (cont.)



JTextField: hello

JPasswordField: ******

JFormattedTextField: 10.5.2004

Type text and then Enter in a field.

This is an editable JTextArea. A text area is a "plain" text component, which means that although it can display text in any font, all of the text is in the same font.

This is an uneditable `JEditorPane`, which was *initialized* with **HTML** text from a URL.

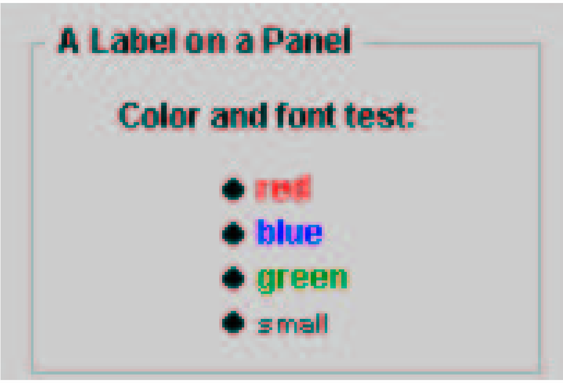
An editor pane uses specialized

JTextPane is a subclass of JEditorPane that uses a StyledEditorKit and StyledDocument, and provides cover

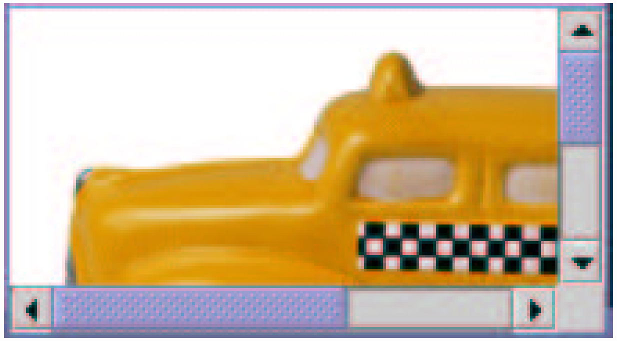
JTextField, JFormattedTextField, JPasswordField
JTextArea, JEditorPane, JTextPane

Containers

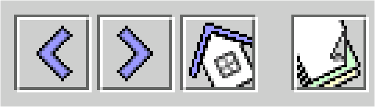
Containers are components that can contain other components.



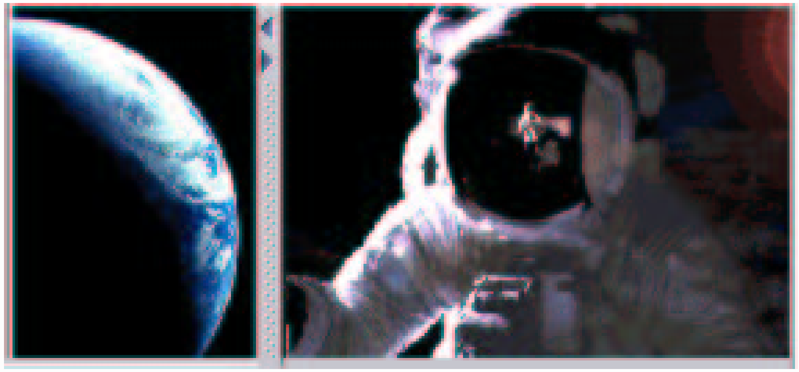
JPanel



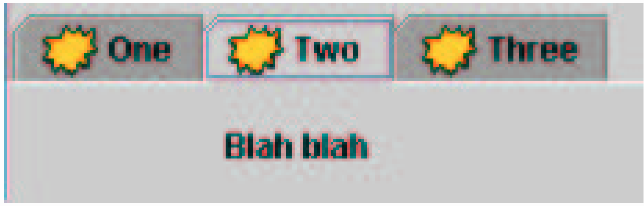
JScrollPane



JToolBar



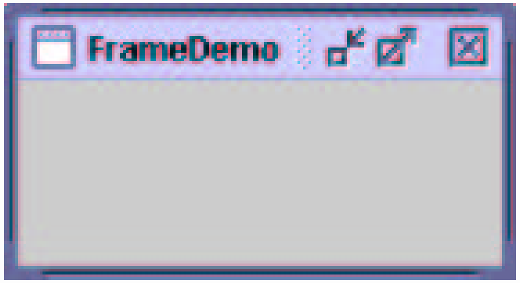
JSplitPane



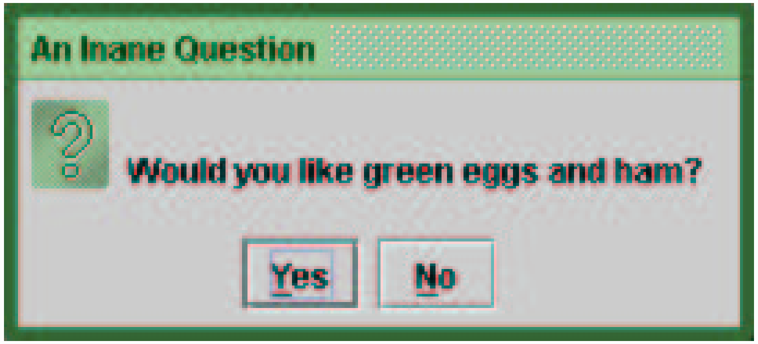
JTabbedPane

Top-Level Containers

Top level containers:



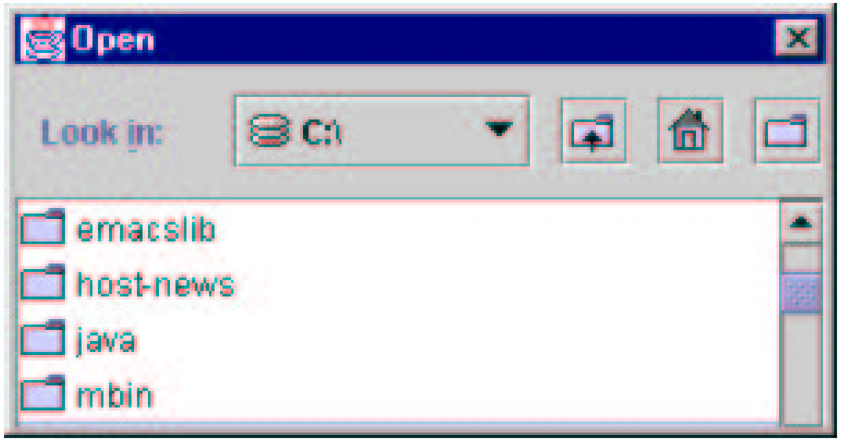
JFrame



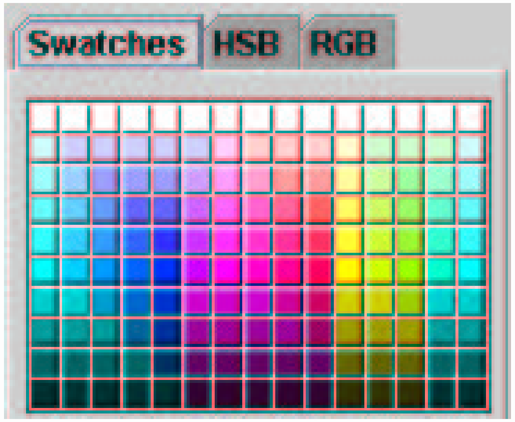
Would you like green eggs and ham?

JDialog

Predefined dialogs:



JFileChooser



JColorChooser

Important Classes from java.awt

Some important classes from the **java.awt** package:

- **Dimension** – width, height
- **Rectangle** – x, y, width, height
- **Insets** – left, right, top, bottom
- **Point** – x, y
- **Color** – e.g, Color.RED, ...
- **Cursor** – e.g., Cursor.WAIT_CURSOR
- **Font**, **FontMetrics**

The JComponent Class

The **JComponent** class is a common superclass of all Swing components.

Overview of methods:

- void setForeground(Color fg), Color getForeground()
- void setBackground(Color bg), Color getBackground()
- void setFont(Font font), Font getFont()
- void setCursor(Cursor cursor), Cursor getCursor(), boolean isCursorSet()
- void setName(String name), String getName()
- void setToolTipText(String text), String getToolTipText()
- void setEnabled(boolean enabled), boolean isEnabled()
- void setVisible(boolean aFlag), boolean isVisible(), boolean isShowing()

The JComponent Class (cont.)

- void repaint(), void repaint(int x, int y, int width, int height), void repaint(Rectangle r)
- void revalidate()
- void paintComponent(Graphics g)
- void setPreferredSize(Dimension preferredSize), Dimension getPreferredSize()
- void setMaximumSize(Dimension maximumSize), Dimension getMaximumSize()
- void setMinimumSize(Dimension minimumSize), Dimension getMinimumSize()
- boolean isMinimumSizeSet(), boolean isPreferredSizeSet(), boolean isMaximumSizeSet()

The JComponent Class (cont.)

- int getWidth(), int getHeight()
- Dimension getSize(), Dimension getSize(Dimension rv)
- int getX(), int getY(), Point getLocation(),
  Point getLocation(Point rv), Point getLocationOnScreen()
- Rectangle getBounds(), Rectangle getBounds(Rectangle rv)
- Insets getInsets(), Insets getInsets(Insets insets)
- void setLocation(int x, int y), void setLocation(Point p)
- void setSize(int width, int height), void setSize(Dimension d)
- void setBounds(int x, int y, int width, int height),
  void setBounds(Rectangle r)
- void setBorder(Border border), Border getBorder()

The JComponent Class (cont.)

Dealing with component hierarchy:

- Component add(Component comp), Component add(Component comp, int index), void add(Component comp, Object constraints), void add(Component comp, Object constraints, int index)
- void remove(int index), void remove(Component comp), void removeAll()
- Component getComponent(int n), Component[] getComponents(), int getComponentCount()
- Container getParent(), Container getTopLevelAncestor()

Events and Event Listeners

Events can be divided into two groups:

- **low-level** events - low-level input, window-system occurrences, e.g., KeyEvent, MouseEvent, MouseWheelEvent, PaintEvent, WindowEvent
- **semantic** events - everything else, e.g., ActionEvent, FocusEvent, ItemEvent, TextEvent

Overview of the most important listeners:

- KeyListener, MouseListener, MouseMotionListener, MouseWheelListener
- WindowListener, WindowFocusListener, WindowStateListener
- ActionListener, ChangeListener, ItemListener
- ComponentListener
- MenuListener, MenuKeyListener, PopupMenuListener

Adapter Classes

The adapter class implements its corresponding listener class by providing all of the required methods, but which have bodies that do nothing.

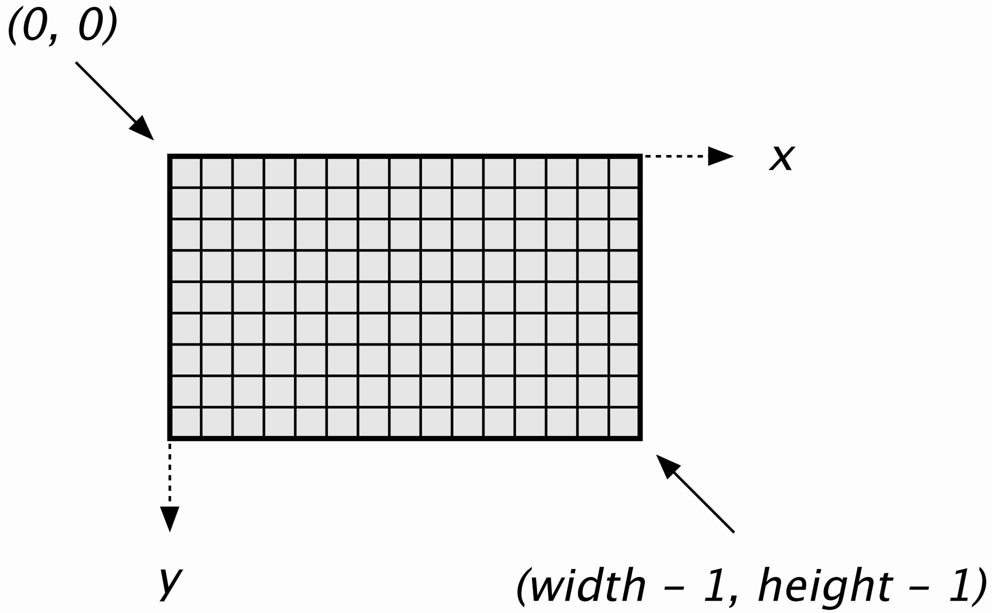
```
public abstract class MouseMotionAdapter
      implements MouseMotionListener {

   public void mouseDragged(MouseEvent e) {
   }

   public void mouseMoved(MouseEvent e) {
   }
}
```

The adapter classes serve as base classes for event handlers.

```
addMouseMotionListener(new MouseMotionAdapter() {
   public void mouseMoved(MouseEvent e) {
      ...
   }
});
```

Graphics Programming

Each component has its own integer coordinate system, ranging from *(0,0)* to *(width–1, height–1)*, with each unit representing the size of one pixel. The upper left corner of a component's painting area is *(0,0)*. The *x* coordinate increases to the right, and the *y* coordinate increases downward.



Note: A pixel is a “picture element”. It is a dot on a computer screen.

Painting

Painting mechanism schedules painting of visible components. It takes care of details such as damage detection, clip calculation and *z*-ordering. There are two kinds of painting operations.

- **System-triggered**
  The system requests a component to render its contents, usually for one of the following reasons:
  - the component is first made visible on the screen,
  - the component is resized,
  - the component has damaged that needs to be repaired.
- **Application-triggered**
  A component decides it needs to update its contents because its internal state has changed. An application invokes `repaint()` method on a component, which registers an **asynchronous** request that this component needs to be repainted. The component is then repainted by invocation of `paintComponent()` method.

  Note: If multiple calls to `repaint()` occur on a component before the initial repaint request is processed, the multiple requests may be collapsed into a single call to `paintComponent()`.

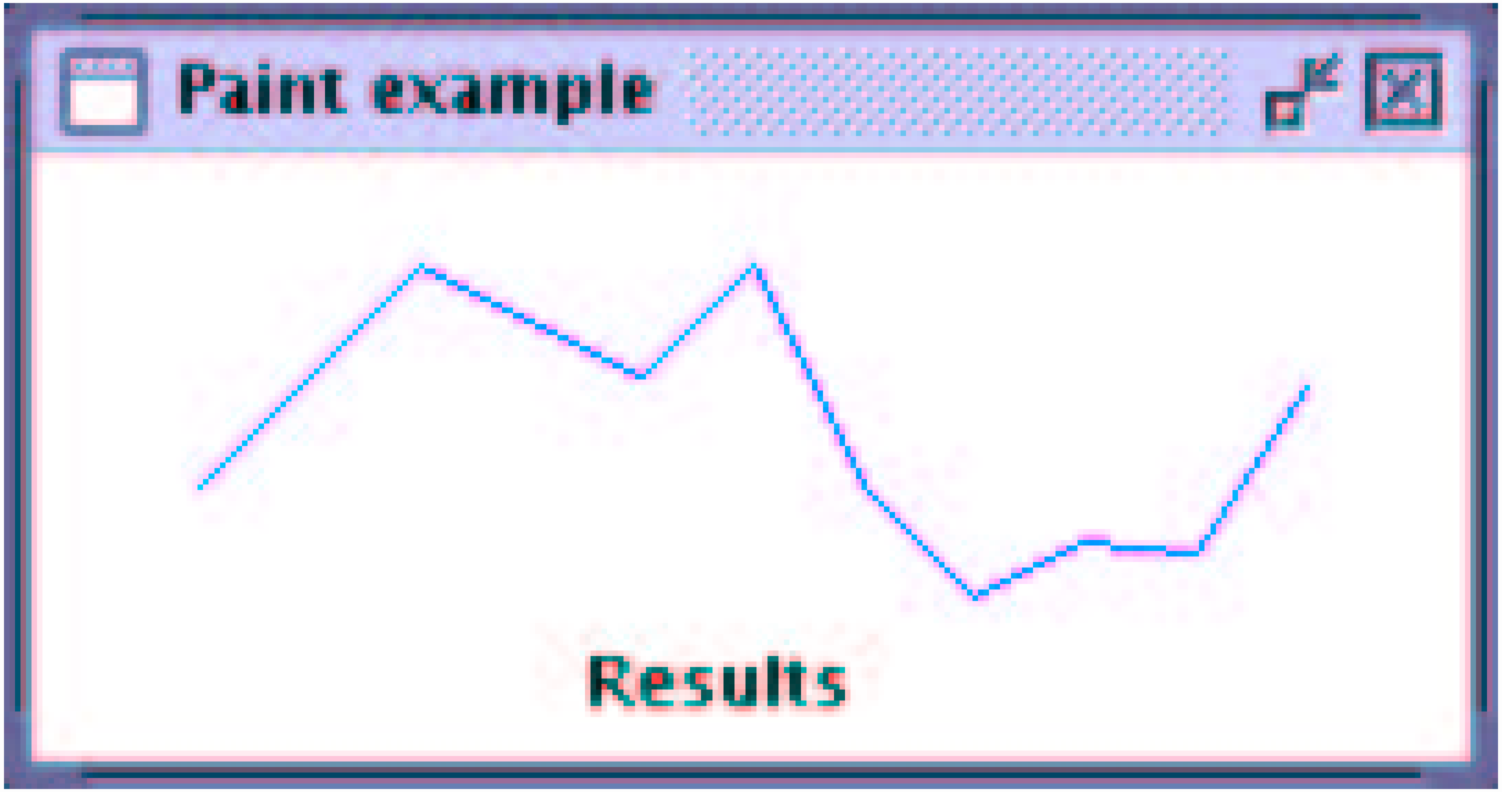
The Graphics Object

The `Graphics` object provides both a context for painting and methods for performing the painting, for example `drawLine()`, `drawRect()`, `fillRect()`, `drawPolygon()`, `drawString()`, etc. It encapsulates information needed for the basic rendering operations. The information includes the following properties.

- a component on which to draw,
- a translation origin for rendering and clipping coordinates,
- current clip,
- current color,
- current font,
- ...

Note: The `Graphics` objects take up operating system resources (more than just memory) and a window system may have a limited number of them.

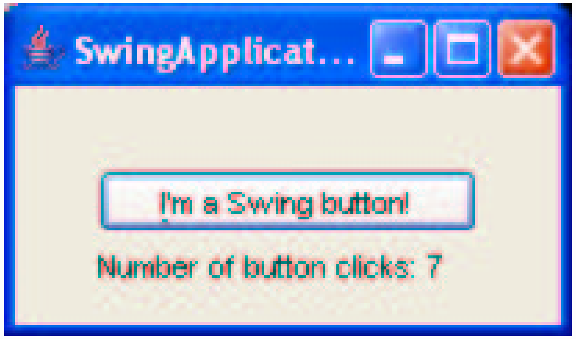
Painting Example



Painting Example (cont.)

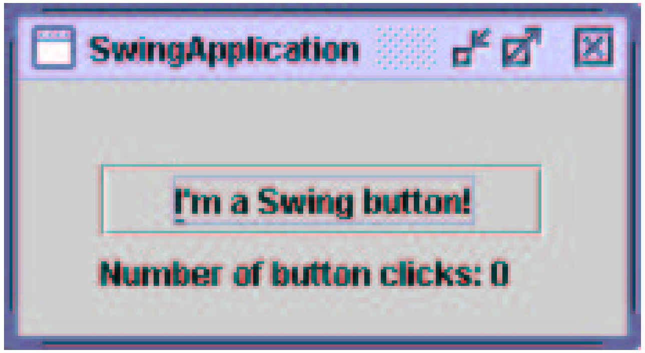
```
public class Graph extends JComponent {
   private int[] values = {
      60, 40, 20, 30, 40, 20, 60, 80, 70, 72, 42
   };

   public Dimension getPreferredSize() {
      return new Dimension(260, 110);
   }

   public void paintComponent(Graphics g) {
      g.setColor(Color.BLUE);
      int step = 20;
      int x = 10;
      for (int i = 0; i < values.length-1; i++) {
         x += step;
         g.drawLine(x, values[i], x+step, values[i+1]);
      }
      g.setFont(new Font("sans", Font.BOLD, 12));
      g.setColor(Color.BLACK);
      g.drawString("Results", 100, 100);
   }
}
```

Look and Feel



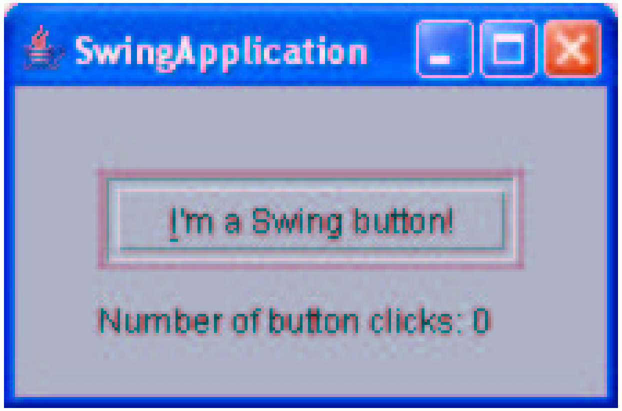
Number of button clicks: 7

Windows look and feel



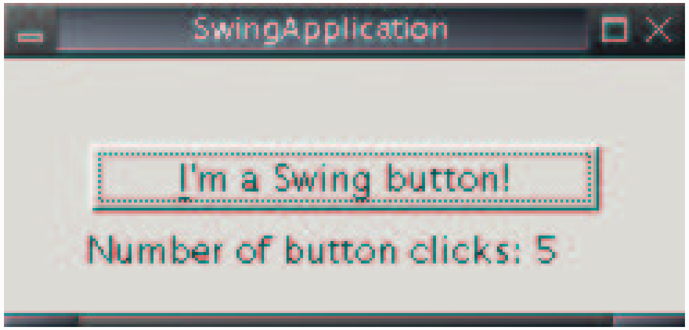
Number of button clicks: 0

Java look and feel



Number of button clicks: 0

Motif look and feel



Number of button clicks: 5

GTK+ look and feel

Look And Feel (cont.)

Setting the look and feel:

```
UIManager.setLookAndFeel(
        UIManager.getCrossPlatformLookAndFeelClassName());
```

Possible values of look and feel argument:

- `UIManager.getCrossPlatformLookAndFeelClassName()`
- `UIManager.getSystemLookAndFeelClassName()`
- `"javax.swing.plaf.metal.MetalLookAndFeel"`
- `"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"`
- `"com.sun.java.swing.plaf.motif.MotifLookAndFeel"`
- `"com.sun.java.swing.plaf.gtk.GTKLookAndFeel"`

Look And Feel (cont.)

It is also possible to specify the look and feel at the command line:

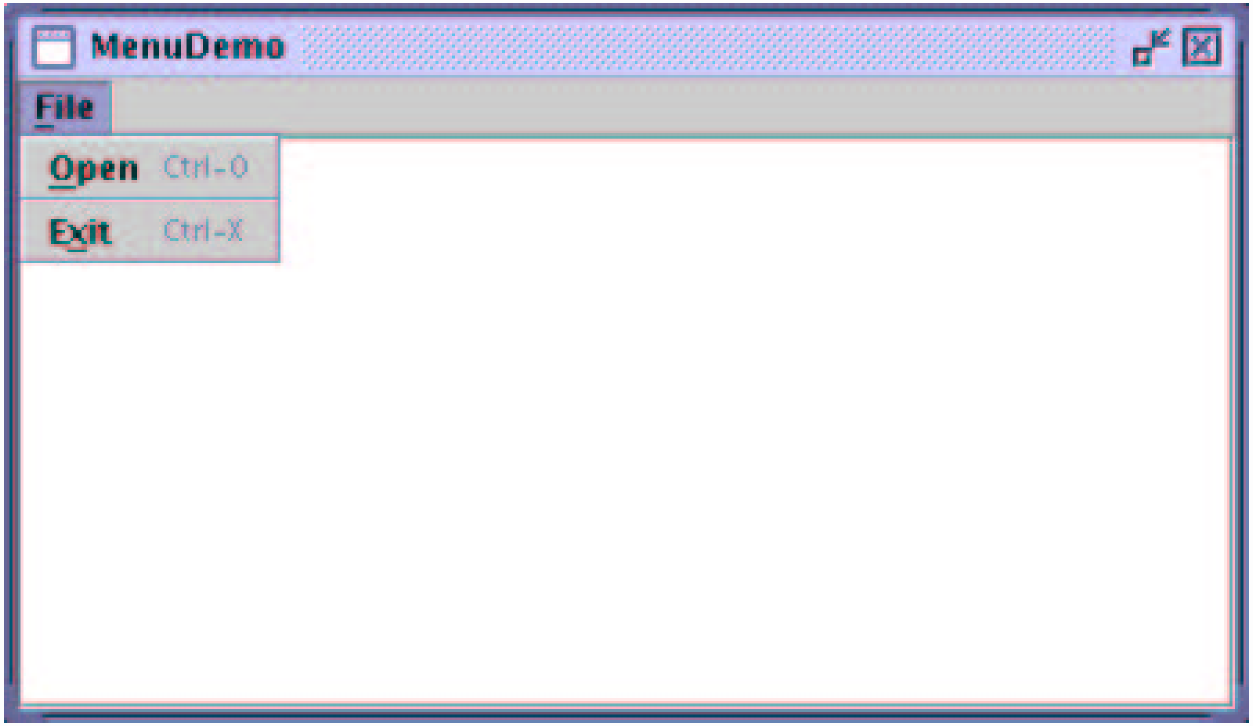
```
$ java -Dswing.defaultlaf=... MyApp
```

Another possibility is to modify the configuration file `swing.properties` in the `jre/lib` directory:

```
# Swing properties

swing.defaultlaf=...
```

Using Menus

The following example illustrates the use of menus:



Using Menus (cont.)

```
private JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();

    JMenu menu = new JMenu("File");
    menu.setMnemonic(KeyEvent.VK_F);
    menuBar.add(menu);

    JMenuItem miOpen = new JMenuItem("Open", KeyEvent.VK_O);
    miOpen.setAccelerator(KeyStroke.getKeyStroke(
            KeyEvent.VK_O, ActionEvent.CTRL_MASK));
    miOpen.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            output.append("Action: Open\n");
        }
    });
    menu.add(miOpen);

    menu.addSeparator();
```

Using Menus (cont.)

```
    JMenuItem miExit = new JMenuItem("Exit", KeyEvent.VK_X);
    miExit.setAccelerator(KeyStroke.getKeyStroke(
            KeyEvent.VK_X, ActionEvent.CTRL_MASK));
    miExit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    });
    menu.add(miExit);

    return menuBar;
}
```

Using Menus (cont.)

```
private Container createContentPane() {
    JPanel contentPane = new JPanel(new BorderLayout());
    contentPane.setOpaque(true);
    output = new JTextArea(5, 30);
    output.setEditable(false);
    JScrollPane scrollPane = new JScrollPane(output);
    contentPane.add(scrollPane, BorderLayout.CENTER);
    return contentPane;
}
```

```
JFrame frame = new JFrame("MenuDemo");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

MenuDemo demo = new MenuDemo();
frame.setJMenuBar(demo.createMenuBar());
frame.setContentPane(demo.createContentPane());

frame.setSize(450, 260);
frame.setVisible(true);
```

Handling Key Events

There are three methods declared in the **KeyListener** interface:

- void keyTyped(KeyEvent e)
- void keyPressed(KeyEvent e)
- void keyReleased(KeyEvent e)

A **KeyEvent** object contains the following information:

- the key code (constants such as VK_A, VK_LEFT, VK_PAGE_DOWN, VK_F5, ...)
- the character associated with the key
- the modifiers – Shift, CTRL, Meta (Alt)
- key location – left, right, standard, numpad
- if it is an action key

Icons

An **icon** is a fixed-sized picture. An icon is an object that implements the **Icon** interface.

The **ImageIcon** class is an implementation of the **Icon** interface that paints icon from a GIF, JPEG, or PNG image.

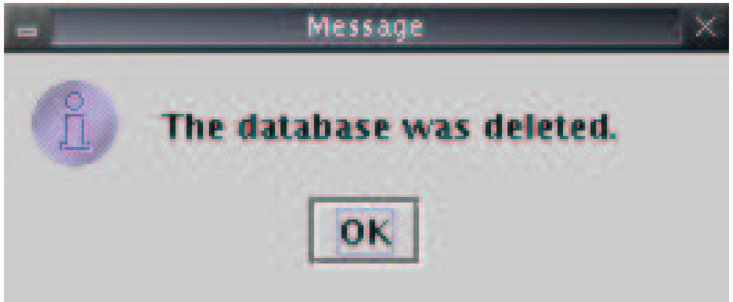
```
protected static Icon createImageIcon(String path) {
    java.net.URL imgURL = IconDemo.class.getResource(path);
    if (imgURL != null) {
        return new ImageIcon(imgURL);
    } else {
        System.err.println("Couldn't find file: " + path);
        return null;
    }
}
```

```
Icon icon = createImageIcon("images/my_image.png");
JLabel label = new JLabel(icon);
```

Simple Dialogs

To create and show simple dialogs, we can use the **JOptionPane** class:

```
JOptionPane.showMessageDialog(frame,
                              "The database was deleted.");
```

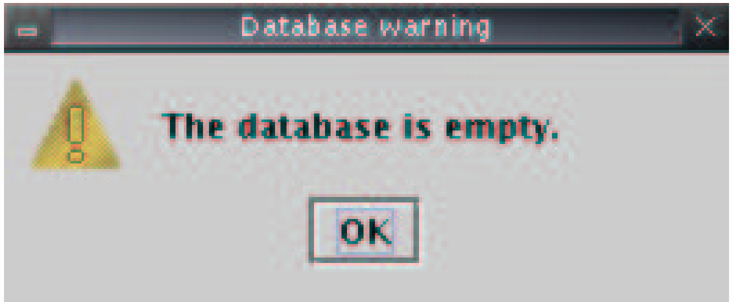


Message

The database was deleted.

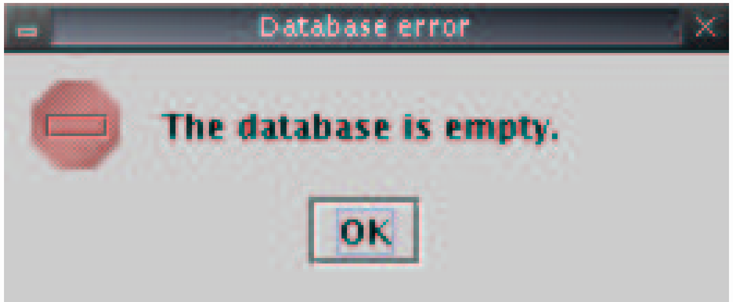
Simple Dialogs (cont.)

```
JOptionPane.showMessageDialog(frame, "The database is empty.",
      "Database warning", JOptionPane.WARNING_MESSAGE);
```



The database is empty.

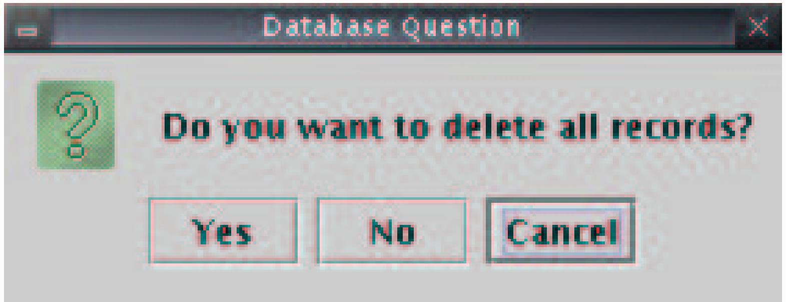
```
JOptionPane.showMessageDialog(frame, "The database is empty.",
      "Database error", JOptionPane.ERROR_MESSAGE);
```



The database is empty.

Simple Dialogs (cont.)

```
Object[] options = { "Yes", "No", "Cancel" };
int n = JOptionPane.showOptionDialog(frame,
        "Do you want to delete all records?",
        "Database Question",
        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, options, options[2]);
```



Database Question

Do you want to delete all records?

Timers

A Swing timer (instance of the **javax.swing.Timer**) fires one or more action events after a specified delay.

Swing timer can be used in two ways:

- to perform a task once, after a delay
- to perform a task repeatedly

```
Timer timer = new Timer(1000, new ActionListener() {
    public void actionPerformed(ActionEvent e) {
          . . .
    }
});

timer.start();
```