Introduction
to the
Java Programming Language

Onur Derin
ALaRI, Faculty of Informatics, USI
derino@alari.ch

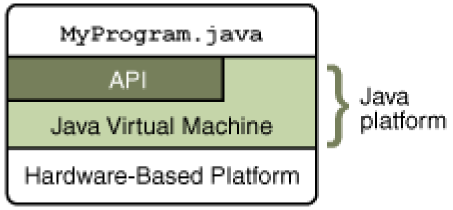
{2.11.2011, 7.11.2011, 24.11.2011}

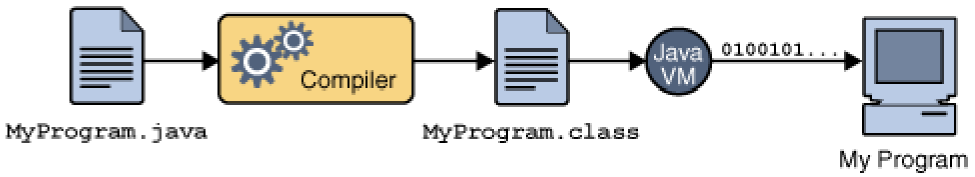
Contents

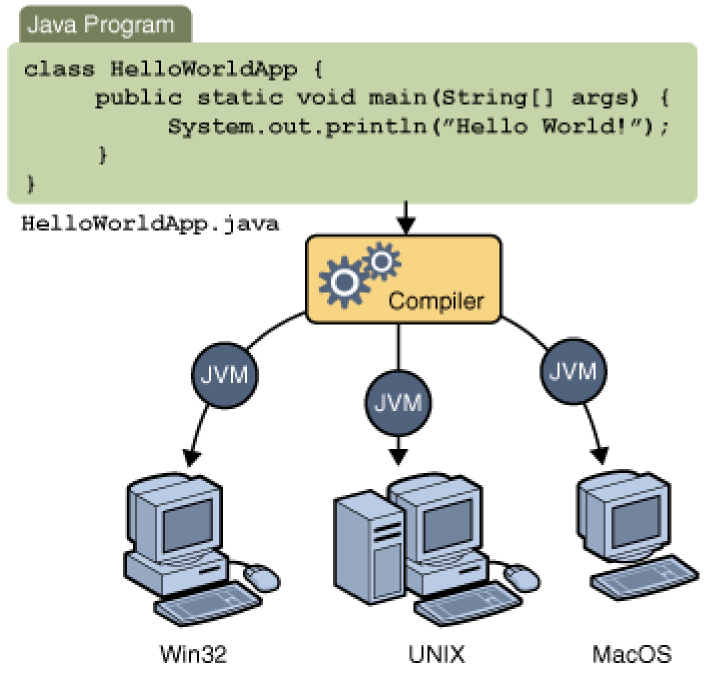
Java Programming Language

- Simple
- Architecture neutral
- Object oriented
- Portable
- Distributed
- High performance (!)
- Multi-threaded
- Robust
- Dynamic
- Secure
- Open source

Java Platform
- Java API
- JVM







```
class HelloWorldApp {
     public static void main(String[] args) {
          System.out.println("Hello World!");
     }
}
```

Java Virtual Machine

- Java is compiled into bytecodes
- Bytecodes are high-level, machine-independent instructions for a hypothetical machine, the Java Virtual Machine (JVM)
- The Java run-time system provides the JVM
- The JVM interprets the bytecodes during program execution
- Since the bytecodes are interpreted, the performance of Java programs slower than comparable C/C++ programs
- But the JVM is continually being improved and new techniques are achieving speeds comparable to native C++ code

Major Java Technologies

- J2SE (Java applications, applets)
- J2EE (servlets)
- J2ME (MIDlets)

They have different runtime environments but they are all programmed with the Java language.

Keywords of Java

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
null
package
private
protected
public
return
short

static
super
switch
synchronized
this
throw
throws
transient
try
void
volatile

Not much different from C/C++

Most of the time doing things the C++ way works

BNF Index of JAVA language grammar:
http://cui.unige.ch/db-research/Enseignement/analyseinfo/JAVA/

Difference of Java from C++

- No typedefs, defines or preprocessor
- No header files
- No structures or unions
- No enums
- No functions - only methods in classes
- No multiple inheritance
- No operator overloading (except “+” for string concatenation)
- No automatic type conversions (except for primitive types)
- No pointers

Naming Conventions

Identifier type	**Convention**	**Examples**
Class names	Capitalize each word within identifier	ConnectionManager
Method names	Capitalize each word except the first	connectPhone
Variable names	Capitalize each word except the first	phoneNumber
Constant MAX_CONNECTIONS	Capitalize each word with underscores btw words	

If you comply with these conventions, you make everyone's life easier.

Structure of a Java Source File

```
// AnExample.java

package ch.alari.javatutoring;

import java.io.*;

class AnExample
{
        private int x;

        public AnExample(int x)
        {
                this.x = x;
        }

        /* Interface of the class
            to other classes.
        */
        public int getX()
        {
                return x;
        }

        public void setX(int x)
        {
                this.x = x;
        }
}
```

Source file should have the same name as the class it declares

AnExample.java needs to reside in ch/alari/javatutoring

Imported classes need to be in the classpath at compile-time

Variable declarations

Constructor

Method declarations

Information on Exercises

- These slides are accompanied with a **lab-skeletons.tar.gz** file where all the exercises that will be done during the class are put in a directory structure along with
  - a **test driver class** for the classes you are expected to write,
  - **Makefile**s and
  - text files that show the **correct output** of programs.

- **make** to compile
- **make run** to run
- **make check** to see if your implementation is correct
- **make clean** to remove *.class files

- See the **README** file in lab-skeletons.tar.gz for more information

HelloWorld Example (ex.1)

```
// HelloWorld.java

class HelloWorld
{
        public HelloWorld()
        {
        }

        public static void main(String[] args)
        {
                System.out.println(“Hello World!”);
        }
}
```

Compiling

```
export PATH=$PATH:/opt/java/bin
javac HelloWorld.java
```

Running

```
java HelloWorld
```

HelloWorld Example using packages (ex.2)

```
// HelloWorld.java

package ch.alari.javatutoring.examples;

class HelloWorld
{
	public HelloWorld()
	{
	}

	public static void main(String[] args)
	{
		System.out.println(“Hello World!”);
	}
}
```

Compiling

```
javac
ch/alari/javatutoring/examples/HelloWorld.java
```

Running

```
java ch.alari.javatutoring.examples.HelloWorld
```

These commands are issued from within the directory that contains the directory named “ch”!!!

If you import 3rd party classes, you need to include them with “-classpath” argument.

Some Java features

- Garbage collection
- Primitive types: int, double, long, float, boolean
- Anything else is derived from java.lang.Object
  (as if public class MyExampleClass extends Object)
- Every variable in Java (except primitives) is like a reference in C++
- Arguments in method calls are passed always by value. (value of reference)

Pass-by-value(-of-reference) (ex.3)

```
public class Main {

    public Main() {
    }

    public static void main(String[] args)
    {
        Point p1 = new Point(1,2);

        translateBy5(p1);

        System.out.println(p1.x + " " + p1.y);
    }

    public static void translateBy5(Point p)
    {
        p.x += 5;
        p.y += 5;
    }

}
```

Outputs **p1.x=6, p1.y=7**

- Analyze JavaValueReference.java

Lab exercise (ex.4)

- Write a **swap** function that swaps two integers.
- Compile and run your program.
- Check if it works.

  Hint:

  Sol.1) Use an array
  Sol.2) Define a MyInteger class

Lab exercise (ex.5)

- Convert one of the C++ exercises you have written last week into Java.
- Or convert bubblesort.cpp to BubbleSort.java
- Compile and run your program.
- Check if it works.

Object-Oriented Principles with Java

- **Encapsulation**: provided by **private**-**protected**-**public** identifiers in class members

- **Inheritance**:
  - provided by **extends** keyword
  - **abstract** keyword declares methods and classes as abstract
  - **super()**: explicit call for constructing parent

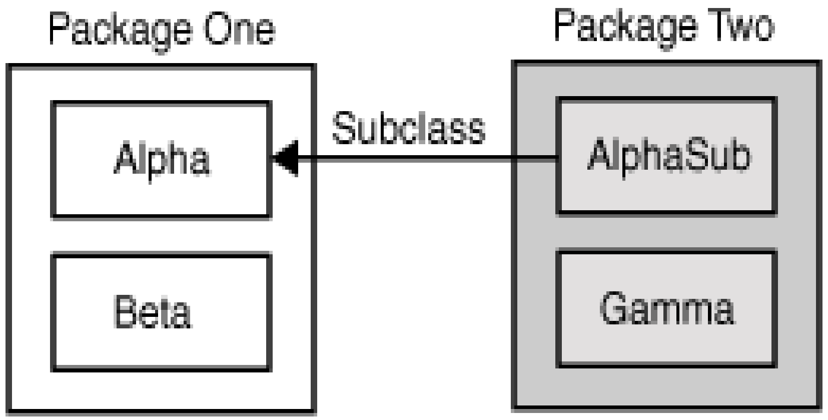
- **Polymorphism**:
  - Method overloading: same method name with different signatures
  - Interfaces: **interface** and **implements** keyword

Encapsulation

- provided by **private-protected-public** identifiers in class members

```
public class Guardian
{
    private Key prisonKey;
    public Key getPrisonKey(Person requester)
    {
        if(requester.getType() == Person.PRISON_STAFF)
            return prisonKey;
        else
            return null;
    }
}
```

Encapsulation



For members of Alpha class:

Visibility Modifier	Alpha	Beta	AlphaSub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

See http://www.uni-bonn.de/~manfear/javaprotection.php for a nice example

Inheritance

- Inheritance:
  - provided by **extends** keyword
  - **abstract** keyword declares methods and classes as abstract
  - **super()**: explicit call for constructing parent

```
• public abstract class Shape
  {
          protected double area;
          protected String name;
          public abstract double getArea();
          public Shape(String name){ this.name = name; }
  }

• public class Rectangle extends Shape
  {
          private double a = 5, b = 6;

          public Rectangle(String name){ super(name);  }

          public double getArea()
          {
                  area = a*b;
                  return area;
          }
  }

• Shape s = new Rectangle("Rectangle A");     // up-casting done implicitly.
• s.getArea();
```

Only single inheritance allowed

Polymorphism

- **Method overloading**: same method name with different signatures

```
public class Matrix
{
        public Matrix(){ ... }
        public Matrix multiply(Matrix m){ ... }
        public Matrix multiply(double scalar){ ... }
}
```

- **Interfaces**: **interface** and **implements** keyword

```
public interface Swimmer
{
        public static final int a = 10;
        public void swim();
}


public class Triathlete implements Swimmer, Runner, Cyclist
{
        public class Triathlete(){...}

        public void swim(){...}

        public void run(){...}

        public void cycle(){...}
}
```

Seems like multiple inheritance

```
Triathlete t = new Triathlete();

Swimmer s = t;
Runner r = t;
Cyclist c = t;

if ( s instanceof Triathlete)
        Triathlete t1 = (Triathlete) s;
```

An Example: Bondus

- Bondus is a program that bonds people together by informing each other on their current status (Available, Busy, Out of office etc.)
- http://www.alari.ch/~derino/bondus
- Interested students are welcome to join

Class Hierarchy

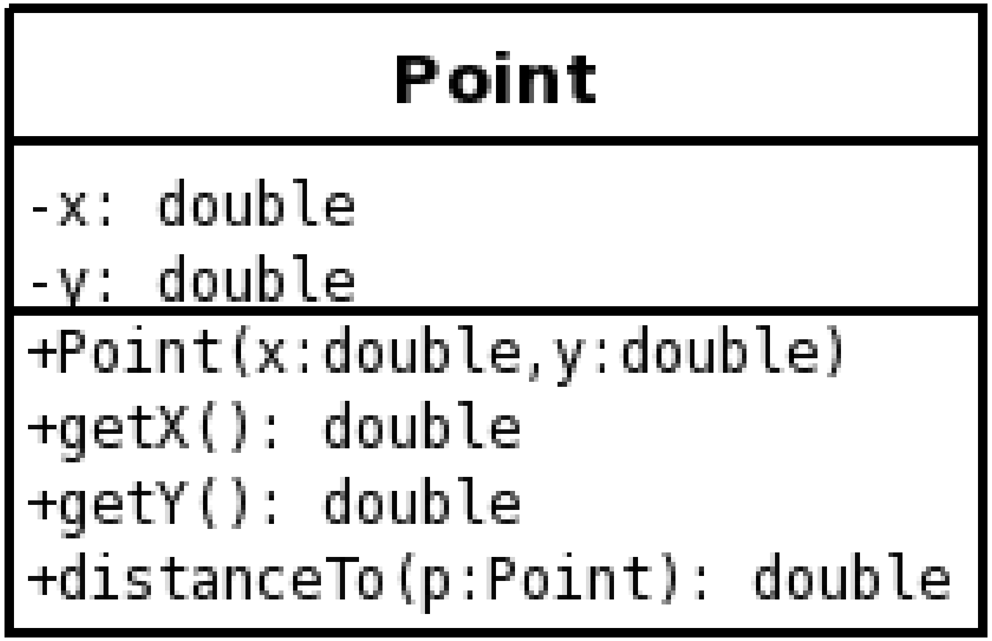
- public interface Publisher
  - public boolean publish(String status);

    - public class FTPPublisher implements Publisher
      - public boolean publish(String status)
        ```
        {
              // connect to FTP server and send the status file
        }
        ```

    - public class SSHPublisher implements Publisher
      - public boolean publish(String status)
        ```
        {
              // connect to SSH server and send the status file
        }
        ```

Lab Exercise (ex.6)

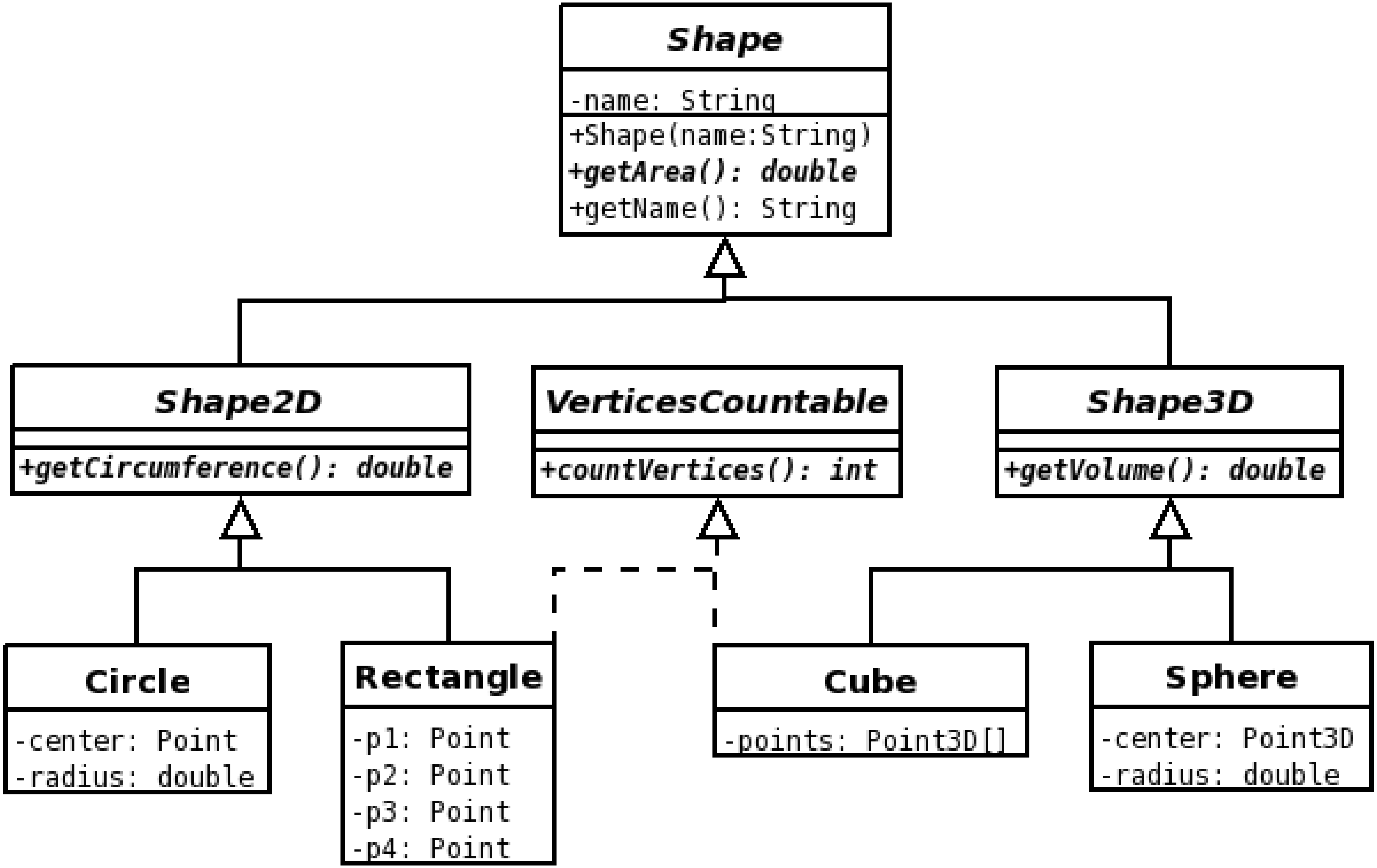
- Given a two dimensional point class (Point.java), extend it to a three dimensional point class (Point3D.java)



```
Point
-x: double
-y: double
+Point(x:double,y:double)
+getX(): double
+getY(): double
+distanceTo(p:Point): double
```

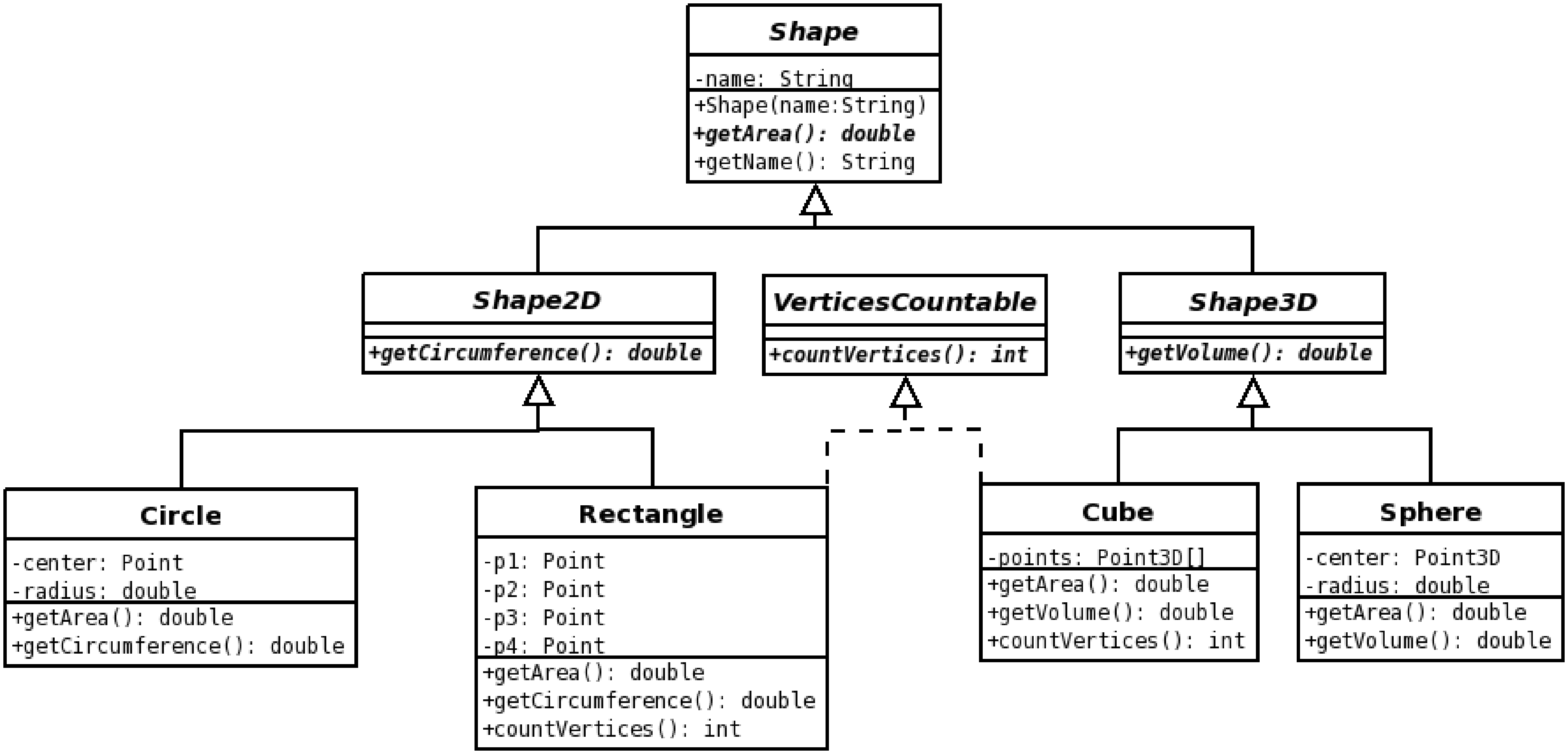
Lab Exercise (ex.7)

- Create the Java code corresponding to the following UML class diagram



Lab Exercise

- Operations of subclasses shown



Exception Handling (ex.8)

- **try, catch, finally, throw, throws, Exception class**

```
public void methodName () throws Aexception, BException
{
      ...
      throw new AException();
      ...
      throw new Bexception();
      ...
 }

 try{
         methodName();          // a code block that can throw an exception.
 } catch(AException ex)
   {
        // handle the exception of type AException
   } catch(BException ex)
   {
        // handle the exception of type BException
   } finally
   {
       // this code block is executed whether there is an exception or not.
   }
```

Exception Handling

- **Define exceptions by extending from Exception class**

```
public class UnauthorizedKeyAccessEx extends Exception
{

}
```

Java API

- All classes contained under the package java and javax.
- import java.* or import javax.*
- Provides a lot of useful classes (Containers, Enumerators, ...)
- Hard to list them all
- Before trying to write a class, first check the Java API.
- Java API is open-source.
- Javadoc documentation of all these classes are available on http://java.sun.com
- Provides a good example of Java programming, useful for self-teaching
- Extensive use of design patterns

3rd Party API's

- A lot of 3rd party APIs are available as open source projects

Iterator Pattern

- Provides a way to access the elements of a collection sequentially without exposing its underlying representation
- Supports multiple, concurrent traversals of collections
- Provides a uniform interface for traversing different collections (that is, supports polymorphic iteration)
- Appears in Java API as

  - **java.util.Iterator:** boolean hasNext(), Object next(), remove()

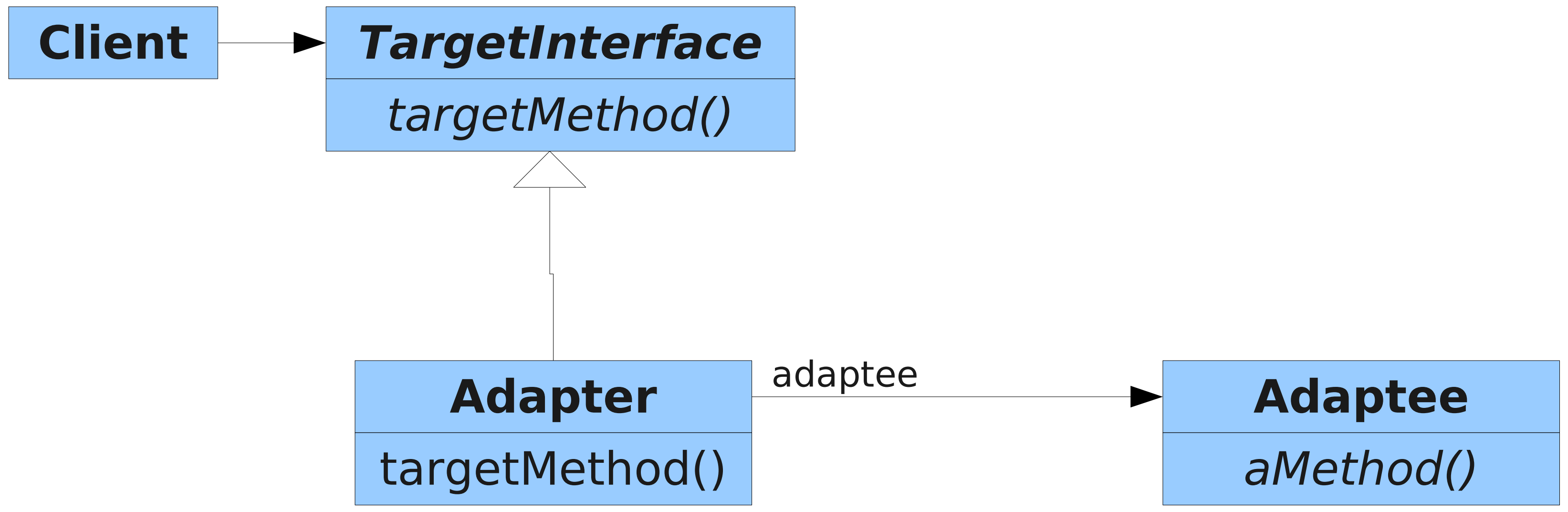
    ```
    for(Iterator i = v.iterator(); i.hasNext(); )
          System.out.println(i.next());
    ```

  - **java.util.Enumeration:** boolean hasMoreElements(), Object nextElement()

    ```
    for (Enumeration e = v.elements(); e.hasMoreElements(); )
          System.out.println(e.nextElement());
    ```

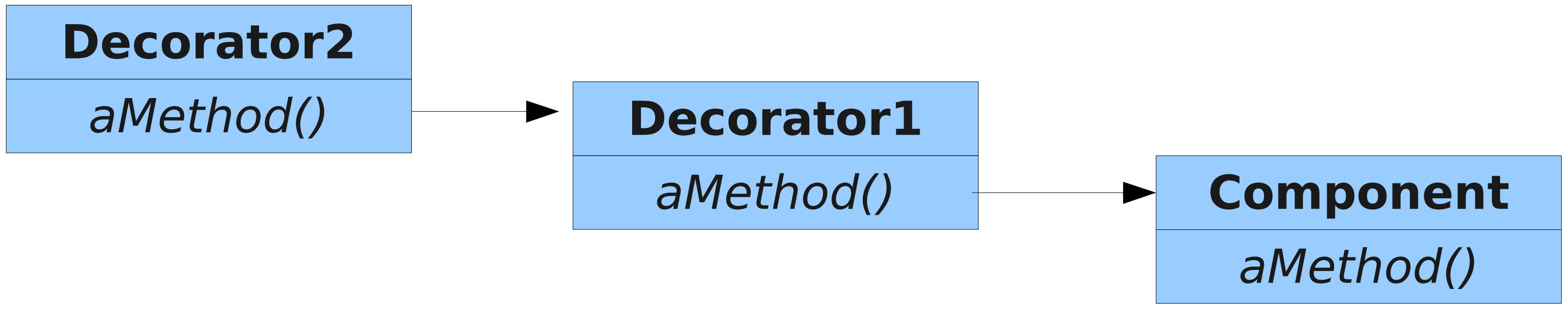
Adapter Pattern

- Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also known as Wrapper
- You want to use an existing class, and its interface does not match the one you need



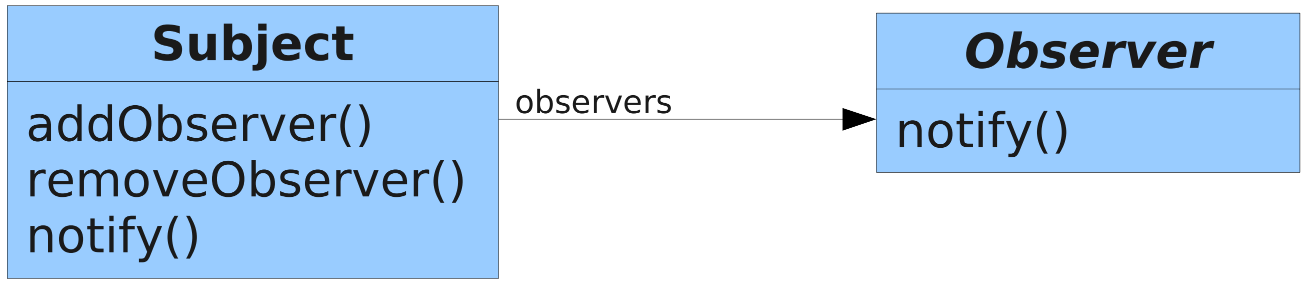
Decorator Pattern

- A flexible alternative to subclassing for extending functionality
- Also known as Wrapper



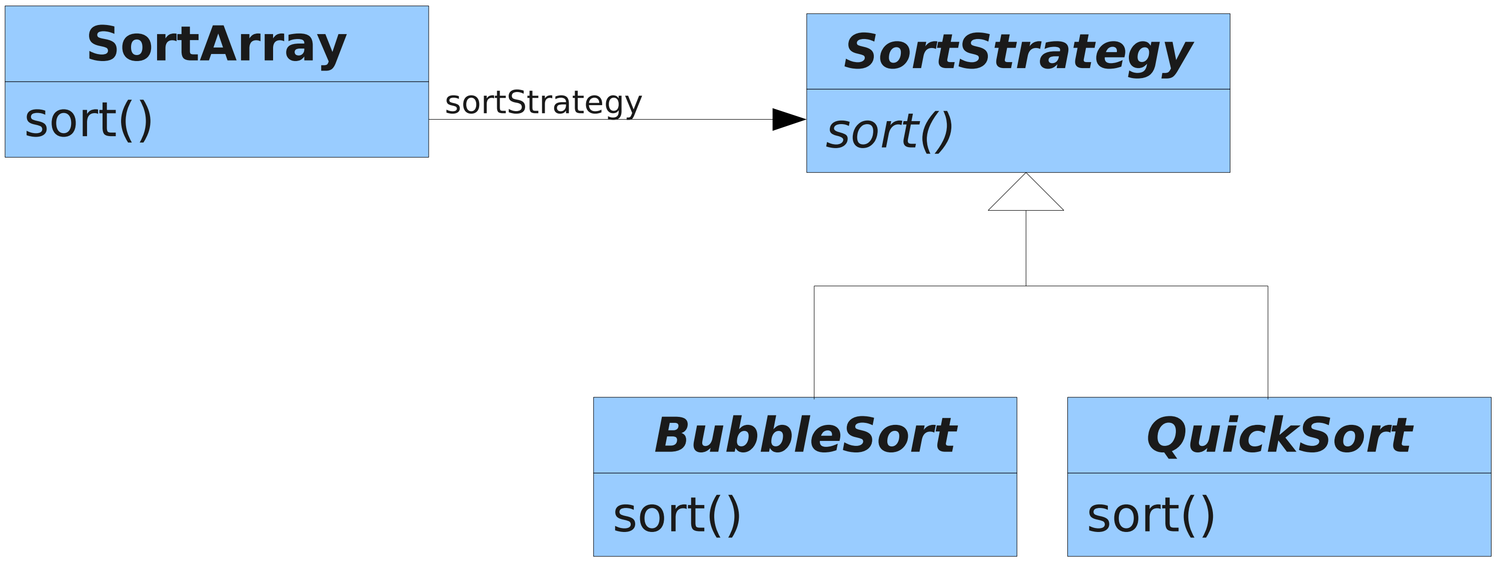
Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically
- Maintains consistency between related objects without making classes tightly coupled
- Corresponds to callbacks/signals in C/C++



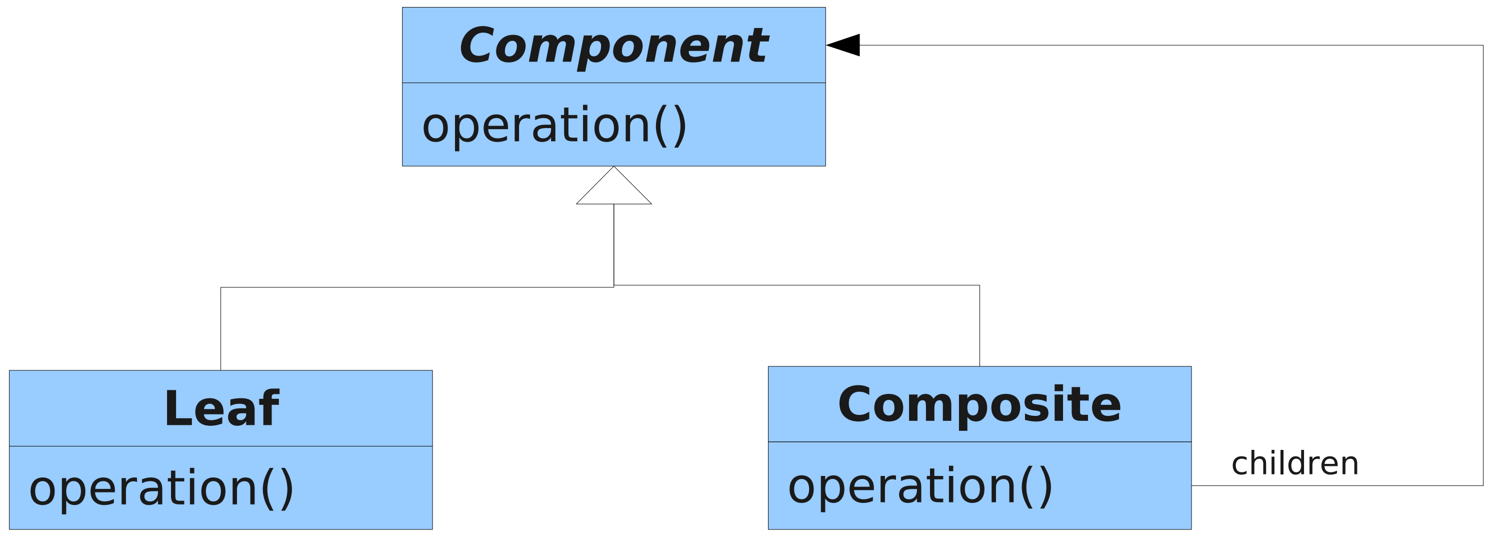
Strategy Pattern

- Define variants of algorithms, encapsulate each one, and make them interchangeable Clients can use different algorithms when needed.
- Provides an alternative to sub-classing to get a variety of algorithms.



Composite Pattern

- Compose objects into three structures
- Individual objects and compositions of objects can be treated uniformly



Abstract Factory Pattern

- In the case when
  - A system should be independent of how its products are created, composed and represented

  - A family of related product objects is designed to be used together and you need to enforce this constraint

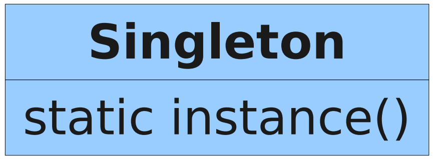
  - e.g. Consider a VLSI design tool. The class to instantiate an AND gate may vary depending on the configuration set by the designer.

    AbstractFactory -> GateFactory (has createAND(), createOR() etc. methods)
    ConcreteFactory -> SiliconGateFactory (Silicon as the underlying substrate)
    ConcreteFactory -> TechnologyGateFactory (45, 60, 90nm technologies)

    AbstractProduct -> ANDGate
    ConcreteProduct -> SiliconBasedANDGate
    ConcreteProduct -> 45nmANDGate

Singleton Pattern

- To make sure there is only one instance of a class and it is accessible globally



```
class Singleton {

      private static Singleton uniqueInstance = new Singleton();
      ...
      private Singleton(){
      ...
      }

      public static Singleton instance() {
          return uniqueInstance;
      }
      ...
  }
```

To access to the singleton

```
Singleton s = Singleton.instance();
```

Java Collections Framework

- **A collection is an object that represents a group of objects**

- **Using this framework**
  - Reduces programming effort
  - Increases performance
  - Provides interoperability between unrelated APIs
  - Reduces the effort required to learn APIs
  - Reduces the effort required to design and implement APIs
  - Fosters software reuse

- It consists of
  - **Collection interfaces** (Collection, Set, List, Map, Queue, …)
    - **Set**: No duplicate elements permitted. May or may not be ordered.
    - **List**: A sequence. Duplicates are generally permitted. Allows positional access.
    - **Map**: A mapping from keys to values. Each key can map to at most one value.
  - **Several implementations** which differ in abstraction and performance criteria

  - **Algorithms** that perform useful functions on collections

  - **Infrastructure interfaces** (Iterator and Enumeration)

java.util.Vector

- Is a **List**
- implements a growable array of **object**s.
- Like an array, it contains components that can be accessed using an integer **index**.
- However, the size of a Vector can grow or shrink as needed to accommodate **adding** and **removing** items after the Vector has been created.

Ex.9

```
Vector fruits = new Vector();
fruits.add(“apple”);
fruits.add(“orange”);
fruits.add(“potato”);
fruits.remove(“potato”);
fruits.contains(“potato”);
fruits.size();
fruits.elementAt(0);

for(java.util.Enumeration e= fruits.elements(); e.hasMoreElements(); )
		{
			System.out.println(e.nextElement());
		}
```

Run this code and change it to use generics

Vector exercise (ex.10)

•Using **java.util.Vector** class and **Adapter** pattern, write a **SortedVector** class that implements a **SortedCollection** interface that has methods:

- void addSorted(**Comparable** obj)
- Object elementAt(int index)
- Enumeration elements()
- int size()

•
```
public class SortedVectorTest {
    public SortedVectorTest() {}

   public static void main(String[] args)
   {
       SortedCollection sv = new SortedVector();
       sv.addSorted("g");
       sv.addSorted("k");
       sv.addSorted("b");
       sv.addSorted("c");
       sv.addSorted("l");
       sv.addSorted("p");
       sv.addSorted("a");
       sv.addSorted("z");
       sv.addSorted("s");

       for(Enumeration e = sv.elements(); e.hasMoreElements();)
           System.out.println(e.nextElement());
   }

}
```

Should output:

```
a
b
c
g
k
l
p
s
z
```

java.util.Stack

- represents a last-in-first-out (LIFO) stack of objects.
- extends class Vector with five operations that allow a vector to be treated as a stack.
  - The usual **push** and **pop** operations are provided,
  - as well as a method to **peek** at the top item on the stack,
  - a method to test for whether the stack **is empty**,
  - a method to **search** the stack for an item and discover how far it is from the top.
- When a stack is first created, it contains no items.

Ex.11

```
Stack fruits = new Stack();
fruits.push(“apple”);
fruits.push(“orange”);
fruits.push(“potato”);
fruits.pop();
fruits.search(“potato”);
fruits.isEmpty();
for(Enumeration e= fruits.elements(); e.hasMoreElements(); )
{
    System.out.println(e.nextElement());
}
```

<u>Run this code and change it to use generics</u>

Stack Exercise (ex.12)

Use **java.util.Stack** class for the following exercise.

Write a class that converts a String given in prefix notation form into infix notation form.

Prefix Notation	Infix Notation
-/+*abcde	((((a*b)+c)/d)-e)
/-ab*c+de	((a-b)/(c*(d+e)))
/a+b*c-de	(a/(b+(c*(d-e))))

java.util.Hashtable

- This class implements a hashtable, which maps keys to values.
- Any non-null object can be used as a key or as a value.

Ex.13

```
Hashtable phoneBook = new Hashtable();
phoneBook.put("A", "+41111111111");
phoneBook.put("B", "+41222222222");
phoneBook.put("C", "+41333333333");

if(phoneBook.containsKey("D"))
      phoneBook.remove("D");

for(java.util.Enumeration e= phoneBook.keys(); e.hasMoreElements(); )
{
      Object key = e.nextElement();
      System.out.println(key + ": " + phoneBook.get(key) );
}
```

Run this code and change it to use generics

Hashtable Exercises

Ex.14

- Write a **Dictionary** class by which you can
    - Add words and their definitions
    - Retrieve a specified word
    - Print all words in the dictionary

Ex.15

- Using **java.util.Hashtable** class, write a **HashtableAdapter** class that implements the **OrderedHashtable** interface that has methods
    - void put(Object key, Object value)
    - Object get(Object key)
    - Enumeration keysInPutOrder()

Algorithms for Collection classes

- java.util.Collections class provides static methods that operate on collections.

- For example,

  - static void **sort**(List list)
    - Sorts the specified list into ascending order, according to the natural ordering of its elements.
  - static void **sort**(List list, Comparator c)
    - Sorts the specified list according to the order induced by the specified comparator.

- **Strategy** pattern

- Similar to sorting, you can use other algorithms (like shuffle, search) of Collections class.

Sort exercise (ex.16)

```
class SortTest {
    private Vector list = new Vector();

    public SortTest() {
        list.addElement(new String("grape"));
        list.addElement(new String("apple"));
        list.addElement(new String("orange"));
        list.addElement(new String("cherry"));
    }

    public void sort() {
        Collections.sort(list);
    }

    public void print() {
        System.out.println(list);
    }

    public static void main(String [] args) {
        SortTest s = new SortTest();
        s.sort();
        s.print();
    }
}
```

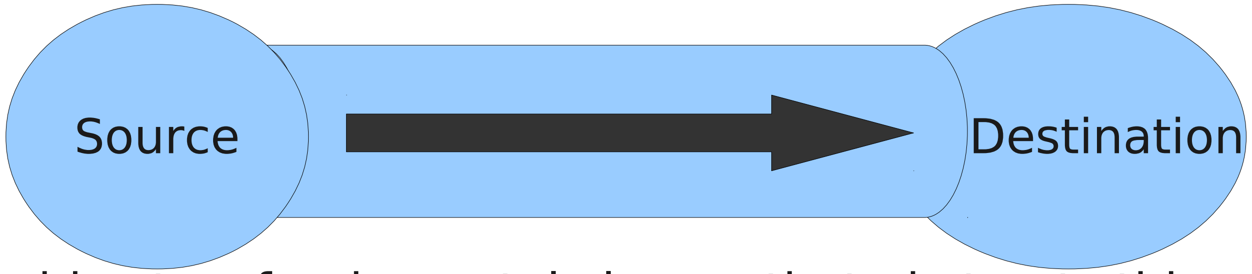
Run this code!

Sort exercise (ex.17)

- Create a Person class with name, age, height fields.
- Using Strategy pattern write AgeComparator, HeightComparator classes deriving from Comparator interface.
- Make a test to sort people according to their age and their height.

Input/Output Operations

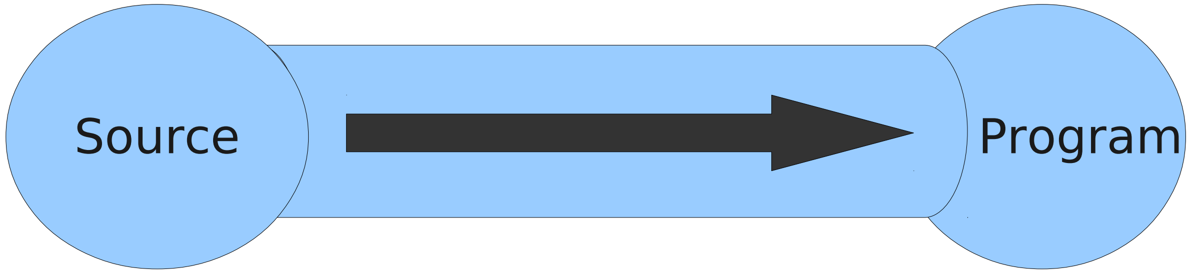
- Fundamental concept is a **stream**.
- Stream: **flow of data from a source to a destination**.



- Java provides two fundamental classes that abstracts this phenomenon.
  - java.io.**InputStream**
  - java.io.**OutputStream**
  - These two classes are byte-oriented.
- Similarly there are
  - java.io.Reader
  - java.io.Writer
  - These are character-oriented.

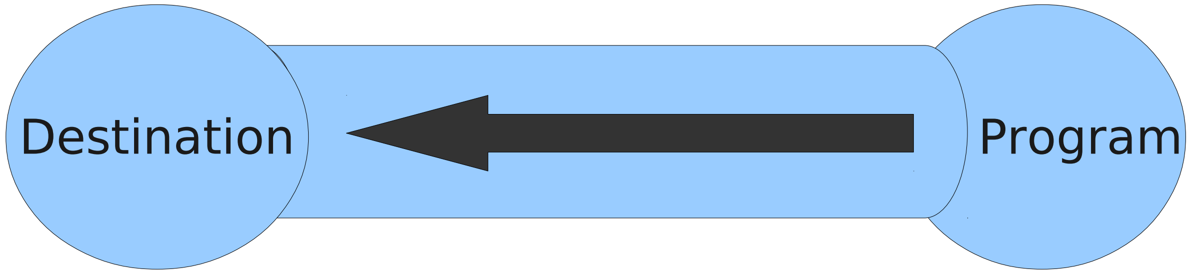
java.io.InputStream

- Is an **abstract** class
- Has abstract **int read()**
- When read is implemented, be careful if it is **blocking**.
- Provides **int available()** to see how many bytes are ready to be read.



java.io.OutputStream

- Is an **abstract** class
- Has abstract **write(int)**



File I/O

- java.io.**FileInputStream**
- java.io.**FileOutputStream**

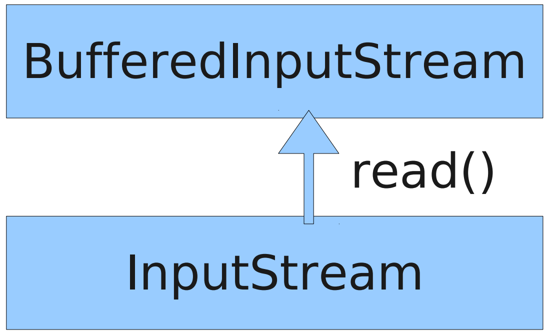
- **Ex. 18**
  Understand the code in Copy.java, compile and run the application.

- As you have noticed, for every byte of the file, there is a function call to **read** which accesses the disk every time it is called.
- Similarly for **write**.

- How to avoid this?
- java.io.**BufferedInputStream**
- java.io.**BufferedOutputStream**

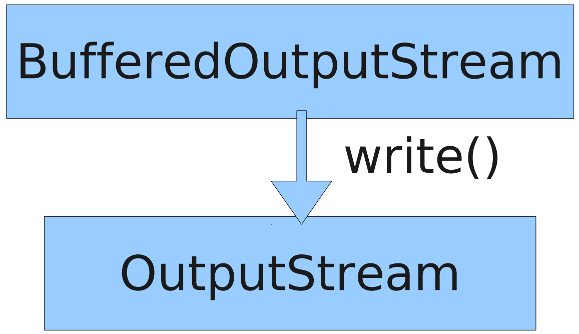
java.io.BufferedInputStream

- Constructed from an InputStream object.
- Holds inside a **buffer** as an array.
- Acts as a **decorator** (remember previous lecture)
- Supports **mark**ing and **reset**ing



java.io.BufferedOutputStream

- Constructed from an OutputStream object.
- Holds inside a **buffer** as an array.
- Acts as an **decorator** (remember previous lecture)
- **write()** is called when buffer is full.



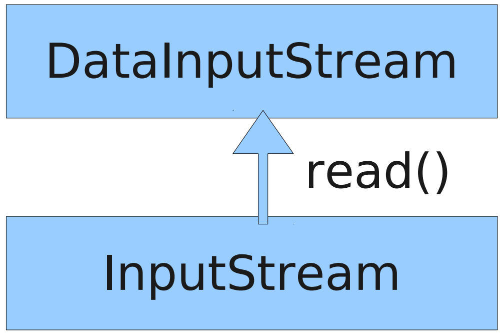
Buffered File I/O

- **Ex. 19**
  - Understand the code in BufferedCopy.java
    (Note constructions of the Buffered IO stream objects!)
  - Compile and run the application.
  - Test results for a file ~4.5Mb

```
[onur@karga src]$ time java Copy test.mp3 test2.mp3

real    0m51.405s
user     0m11.245s
sys     0m35.818s

[onur@karga src]$ time java BufferedCopy test.mp3 test3.mp3

real    0m0.837s
user     0m0.620s
sys     0m0.052s
```

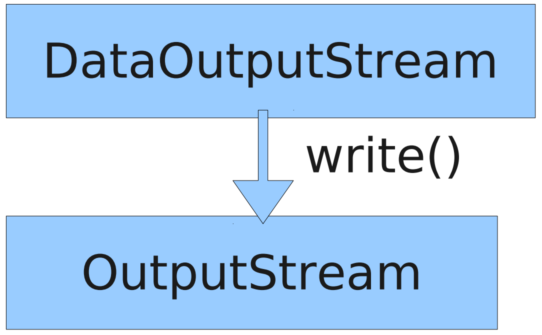
java.io.DataInputStream

- Constructed from an InputStream object.
- Allows reading primitive Java data types from the underlying InputStream object.
- Works in a machine-independent way.
- Acts as a **decorator**



java.io.DataOutputStream

- Constructed from an OutputStream object.
- Allows writing primitive Java data types to the underlying OutputStream object.
- Machine-independent, DataInputStream can be used to read what has been written
- Acts as a **decorator**



Data IO (ex.20)

- Using DataInputStream, DataOutputStream, FileInputStream, FileOutputStream classes,
    - Write an application in which you write an integer, a double and a String in a file.
    - Write an application in which you retrieve what you have written from the file.

A Short Introduction to Multi-threaded Programming in Java

- **Thread:** is a thread of execution in a program.

- There are two ways two make a class run as a Thread.

  1) Extending from **Thread** class and overriding the **run()** method.

```
public class MyClass extends Thread {
...
    public void run()
    {
        ...
    }
}
```

Somewhereelse in your code

```
MyClass m = new MyClass();
m.start();
```

A Short Introduction to Multi-threaded Programming in Java

2) Implementing **Runnable** interface and providing the Runnable object as an argument to the constructor of **Thread.**

```
public class MyClass implements Runnable {
...
    public void run()
    {
        ...
    }
}
```

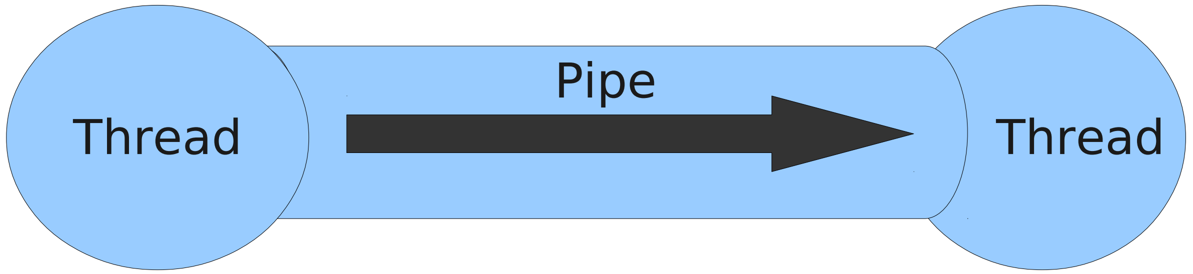
Somewherelse in your code

```
    MyClass m = new MyClass();
    Thread t = new Thread(m);
    t.start();
```

Piped I/O Streams

- java.io.**PipedInputStream**
- java.io.**PipedOutputStream**

- A way for inter-thread communication



- **Ex.21**
  - Understand the code in PipeTest.java,
  - compile and run the application.

GUI Design in Java

- Extensive use of design patterns

- Strategy
  - Container's layout manager (FlowLayout, BorderLayout, GridbagLayout)
  - TextComponent's validator (Numeric, AlphaNumeric, TelNumber)

- Composite
  - See java.awt.Container extends java.awt.Component. Attention to paint() method.

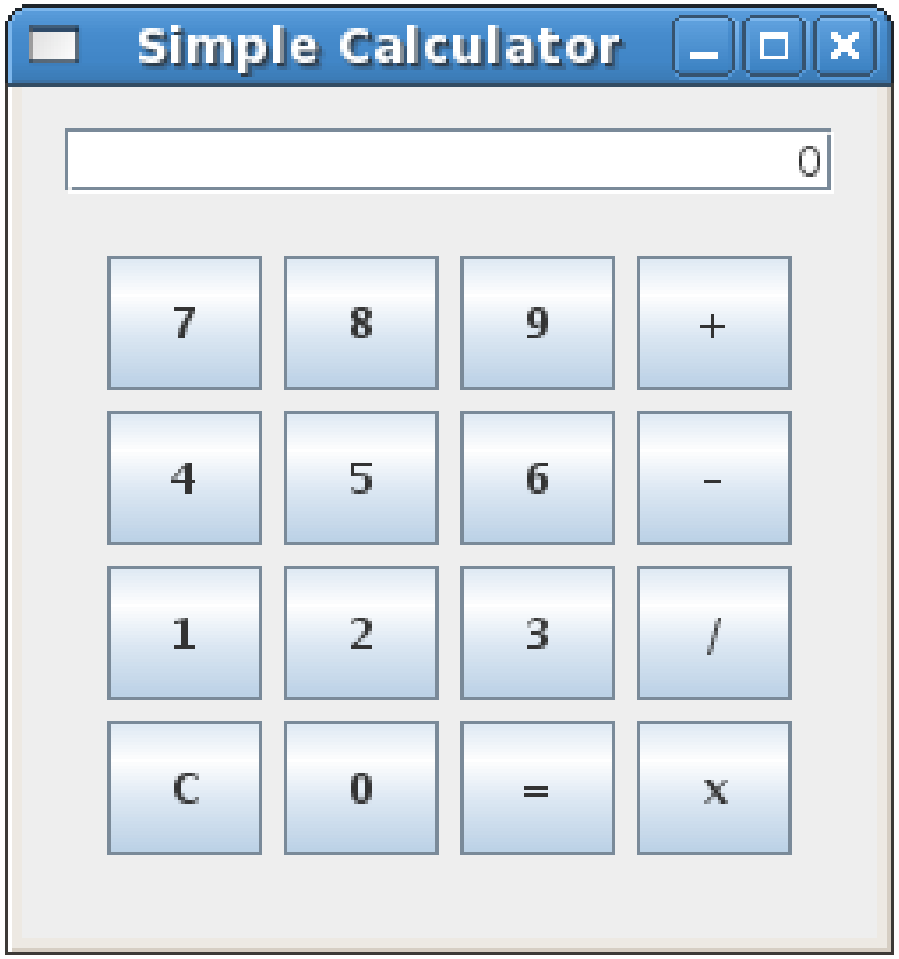
- Observer
  - See java.util.EventListener and all of its subclasses

- Abstract Factory
  - See java.awt.Toolkit and all of its create methods

- Singleton
  - e.g. Toolkit instance is never created by the programmers, see java.awt.Component's getToolkit() method

GUI Design in Java (ex.22)

- Let's see things in action and create a simple calculator



Using external libraries (ex.23) “XML processing with dom4j”

```
import java.io.FileWriter;
import org.dom4j.*;
import org.dom4j.io.*;
import java.util.List;

public class XMLTest {
    Element modelRoot;
    public XMLTest() {}

    public static void main(String[] args){
        Document document = DocumentHelper.createDocument();
        Element projectRoot = document.addElement( "Automata");
        Element automataRoot = projectRoot.addElement("Automaton")
                                          .addAttribute("name", "preg1");

        Element eventsElement = automataRoot.addElement("Events");

        String[] events = {"e0", "e1", "e2"};
        Element modelElement = null;
        for(int i = 0; i<events.length; i++){
                modelElement = eventsElement.addElement("Event");
                modelElement.addAttribute("id",  events[i] );
            }
      }
```

Using external libraries “XML processing with dom4j”

```
// lets write to a file
try
{
    XMLWriter writer = new XMLWriter( new FileWriter("test1.xml") );
    writer.write( document );
    writer.close();
}
catch(Exception e)
{
    System.out.println("exception");
}
```

Using external libraries
“XML processing with dom4j”

```
        Element newElement = (Element)modelElement.clone();
        newElement.addAttribute("id", "e3");
        List eventsList = eventsElement.content();
        eventsList.add(3, newElement);


        // lets write to a file
        try
        {
            XMLWriter writer = new XMLWriter( new FileWriter("test2.xml") );
            writer.write( document );
            writer.close();
        }
        catch(Exception e)
        {
            System.out.println("exception");
        }
    }
```

Compiling: `javac -classpath ./dom4j-1.6.1.jar XMLTest.java`
Running: `java -classpath ./dom4j-1.6.1.jar:. XMLTest`

You can obtain dom4j.jar from www.dom4j.org

Writing an intelligent Pishti player (ex. 24)

- To see the assignment, visit:

  http://www.alari.ch/people/derino/Teaching/Java/Pishti/index.php

ALaRI CfP tracker monthly timeline (ex.25)

- ALaRI CfP tracker lists open and closed call for papers at the following address: http://www.alari.ch/NewsAndEvents/cfp

- Using the ALaRI call for papers XML files

  - wget http://www.alari.ch/NewsAndEvents/cfp/es/cfp.xml
  - wget http://www.alari.ch/NewsAndEvents/cfp/es/cfp-past.xml

  write a program that lists the CfPs grouped by month in a single year timeline according to their submission due dates.
  e.g.
  January
  ABC'08
  XYZ'10
  ABC'09
  ABC'10
  DEF'09
  ABC'11

  February
  ...

ALaRI CfP tracker in webcal format (ex.26)

- ALaRI CfP tracker lists open and closed call for papers at the following address: http://www.alari.ch/NewsAndEvents/cfp
- Using the ALaRI open call for papers XML file
  - wget http://www.alari.ch/NewsAndEvents/cfp/es/cfp.xml

  write a program that creates a file in the webcal format listing the upcoming CfPs according to their submission due dates.

Propose your own assignment (ex. 27)

- Tell me about your own idea, once approved, do it as your final assignment!

References

- Thinking in Java by Bruce Eckel (available online)
- Sun’s Java Homepage: http://java.sun.com
- Bob Tarr’s Design Patterns Homepage: http://research.umbc.edu/~tarr/dp/dp.html
- Jeff Friesen’s Book: Java 2 By Example, Second Edition