BLM-2562 OBJECT ORIENTED PROGRAMMING – Feb. 2018
Associate Prof. Dr. Mehmet AKTAŞ

GENERAL INFORMATION

SCORING

- 1st midterm: %20, 05/04/2018
- 2nd midterm: %20, 10/05/2018
- Final exam: %30,
- Project: %15,
- Lab: %15

- **Exam dates can be changed (follow www.ce.yildiz.edu.tr)**
- The percentages can be changed slightly
- Do not forget what you have learned from the previous term, otherwise penalty points will be deduced from your scores.

SUGGESTED BOOKS:

- Java Programming:
  - Java How to Program, Harvey M. Deitel & Paul J. Deitel, Prentice-Hall.
    - 7th ed. or newer
  - Core Java 2 Volume I & II, C. S. Horstmann and G. Cornell, Prentice-Hall.
    - 7th ed. or newer
- UML:
  - UML Distilled, 3rd ed. (2003), Martin Fowler, Addison-Wesley.
- You can refer to any other book of your choice as well.

COURSE CONTENTS

- Introduction of the course.
- Primitives, wrappers, parameters.
- Exception handling
- Working with Files and Streams (Serialization).
- Introduction to generic classes using basic data structures (Lists).
- Introduction to generic classes using basic data structures (Maps).
- Typecasting, Enum classes, Inner classes.
- Introduction to Multithreading
- GUI Programming (with Swing)

PRIMITIVES AND WRAPPERS

- Primitive type: One unit of information (non-class).
- Wrapper: A class having one primitive number and some useful methods related with that member.
- Natural numbers in Java (Numbers without fractional parts):

Primitive	**Meaning**	**Range**	**Wrapper**
int	Integer (4 bytes)	Lower: – 2.147.483.648 Higher: + 2.147.483.647	Integer
long	Big integer (8 bytes)	(± 9,22 x 10^{18}) `long natID = 12345678900L;`	Long
short	Small integer (2 bytes)	Lower: –32.768 Higher: +32.767	Short
byte	One byte	Lower: –128 Higher: +127	Byte

PRIMITIVES AND WRAPPERS

- Real numbers in Java (Numbers with fractional parts):

Primitive	Meaning	Range	Wrapper
double	Large real number	(± 1,79 10^{308})	Double
float	Small real number	(± 3,4 10^{38})	Float

- Other primitives:

Primitive	Meaning	Range	Wrapper
char	Karakter	'A'-'Z', 'a'-'z', etc. (UTF-16 encoding)	Character
boolean	Mantıksal	false – true	Boolean

- I would also like to remind you the non-primitive type String soon.

PRIMITIVES AND WRAPPERS

- Operations with primitives:
  - Arithmetic: + - * / %.
    - Remember operator predecence
    - ++, --,
    - ++i and i++ differs: y * ++z, y * z++
    - Shorthands: += –= *= /= %=
    - Check the static methods of java.lang.Math: pow, abs, round, …
  - Binary:
    - Boolean algebra: & | ~ ^ (and or not xor)
    - Shifting: << >>
    - Ex: The rightmost 4th bit of n: (n&8)/8 or (n&(1<<3))>>3

PRIMITIVES AND WRAPPERS

- We use wrappers for their useful methods and in cases where primitives cannot be used.
  - Serialization and map indexes are examples of such cases that we will cover in the later weeks.
- The wrappers reside in the java.lang package
- Some useful methods of class Integer (refer to Java API for further methods and more details)
  - int compareTo(Integer anotherInteger)
  - int intValue()
  - static int parseInt(String s)
  - String toString()
  - static String toString(int i)
  - static Integer valueOf(String s)

```
int ilkel1 = 5, ilkel2 = 7;
Integer sarma1, sarma2;
sarma1 = new Integer( ilkel1 );
sarma2 = new Integer( ilkel2 );
System.out.println("Sonuç1:"+sarma1.compareTo(sarma2)); //-1
```

PRIMITIVES AND WRAPPERS

- I would also like to remind you the non-primitive type String and some of its methods:
  - int compareTo(String anotherString)
  - int compareToIgnoreCase(String str)
  - String concat(String str) //The + operation
  - int indexOf (String str) //-1, 0, …
    - "Selçuk". indexOf ("ç") = 3
  - int indexOf (String str, int fromIndex)
  - int length()
  - String substring(int m, int n)
    - Returns characters between positions m and n-1.
      - Because Java (and indexOf) begins counting from 0.
    - "Selçuk".subString(0,3) = "Sel"
  - String toLowerCase()
  - String toUpperCase()
  - String trim()
- P.S. Problems with understanding the workings of these methods can mean that we have fundamental problems with the previous term's Object Oriented Concepts course.

PRIMITIVES AND WRAPPERS

- More methods of the class String:
  - static String format(String format, Object… args)
    - Format parameter: Similar to C format
    - … denote any number of parameters

    ```
    double pi = (double)22/7;
    System.out.println(String.format("%1.6f", pi));
    ```
  - int lastIndexOf (String str)
  - int lastIndexOf (String str, int fromIndex)
  - String replace(CharSequence target, CharSequence replacement)
    - CharSequence interface provides uniform, read-only access to many different kinds of char sequences.

    ```
    String last = "The Last Ninja's Last Supper";
    String first = last.replace("Last", "First");
    System.out.println(first);
    ```

PRIMITIVES AND WRAPPERS

- The primitive version of Enum classes:
  - Sometimes, a variable should only hold a restricted set of values.
  - For example, you may sell pizza in four sizes: small, medium, large, and extra large
    - Of course, you could encode these sizes as integers 1, 2, 3, 4, or characters S, M, L, and X.
    - But that is an error-prone setup. It is too easy for a variable to hold a wrong value (such as 0 or m).
  - Example:
    - Defining a primitive enum (in Size.java):

```
public enum Size {
    SMALL, MEDIUM, LARGE, EXTRA_LARGE;
}
```

    - Using in code:

```
Size s = Size.MEDIUM;
```

  - In fact, we have defined a class named Size and enforced that only four static instances of that class can be created.
    - You cannot write Size s = Size.Medium or MEDIUM or M …

PRIMITIVES AND METHOD PARAMETERS

- Java uses "call-by-value" calling style when passing primitive parameters to methods.
    - This calling style works in the same way when you pass parameters without pointers in the C/C++ language.
- Java uses "call-by-value-of-references" when passing non-primitive parameters to methods.
    - This calling style is different than the style "call-by-references", i.e. when you pass parameters as pointers in the C/C++ language.
    - Well, this style is very similar to the pointer style, except you cannot change the memory address of object parameters
        - This means that changes to object parameters are permanent, except re-initializing and swapping objects.
- Examine the following code and its output:

PRIMITIVES AND METHOD PARAMETERS

```
package oop00;
public class MethodParametersTest1 {
    private Integer wrapI, wrapJ;
    public void ilkelDuzenle( int x ) { x++; }
    public void sarmalayiciDuzenle( Integer x ) { x++; }
    public void ilkelDegistir( int x, int y ) {
         int temp; temp = x; x = y; y = temp;
    }
    public void sarmalayiciDegistir( Integer x, Integer y ) {
         Integer temp; temp = x; x = y; y = temp;
    }
    public void sarmalayiciDegistirAlt(Integer x, Integer y) {
         Integer temp;
         temp = new Integer(x);
         x = new Integer(y);
         y = new Integer(temp);
    }
    public void swapForReal( ) {
         Integer temp = wrapI; wrapI = wrapJ; wrapJ = temp;
    }
    public static void main(String[] args) {
         MethodParameters test = new MethodParameters();
         test.tryMe();
    }
```

PRIMITIVES AND METHOD PARAMETERS

```
    public void tryMe() {
          int count = 3;
          System.out.println("Before : " + count );
          this.ilkelDuzenle(count);
          System.out.println("After: " + count );

          Integer wrap = new Integer( 5 );
          System.out.println("Before : " + wrap );
          this.sarmalayiciDuzenle(wrap);
          System.out.println("After: " + wrap );

          int count1 = 1, count2 = 2;
          System.out.println("Before : " + count1 + ", " + count2 );
          this.ilkelDegistir(count1, count2);
          System.out.println("After: " + count1 + ", " + count2 );

          Integer wrap1 = new Integer( 1 );
          Integer wrap2 = new Integer( 2 );
          System.out.println("Before : " + wrap1 + ", " + wrap2 );
          this.sarmalayiciDegistir(wrap1, wrap2);
          System.out.println("After: " + wrap1 + ", " + wrap2 );

          System.out.println("Before : " + wrap1 + ", " + wrap2 );
          this.sarmalayiciDegistirAlt(wrap1, wrap2);
          System.out.println("After: " + wrap1 + ", " + wrap2 );

          wrapI = 3; wrapJ = 5;
          System.out.println("Before : " + wrapI + ", " + wrapJ );
          this.swapForReal();
          System.out.println("After: " + wrapI + ", " + wrapJ );
    }
}
```

PRIMITIVES AND METHOD PARAMETERS

- The output:

```
Before : 3
After: 3
Before : 5
After: 5
Before : 1, 2
After: 1, 2
Before : 1, 2
After: 1, 2
Before : 1, 2
After: 1, 2
Before : 3, 5
After: 5, 3
```

PRIMITIVES AND METHOD PARAMETERS

- Examine the following code and its output:

```
package oop00;
public class MethodParametersTest2 {
public void tryMe( ) {
        int x = 1, y = 2;
        System.out.println("Before : " + x + ", " + y );
        int temp;
        temp = x;
        x = y;
        y = temp;
        System.out.println("After: " + x + ", " + y );

        Integer sarma1 = new Integer( 3 );
        Integer sarma2 = new Integer( 5 );
        System.out.println("Before : " + sarma1 + ", " + sarma2 );

        Integer gecici = sarma1;
        sarma1 = sarma2;
        sarma2 = gecici;
        System.out.println("After: " + sarma1 + ", " + sarma2 );
    }
```

PRIMITIVES AND METHOD PARAMETERS

```
    public static void main(String[] args) {
        MethodParametersTest2 test = new MethodParametersTest2();
        test.tryMe();
    }
}
```

- The output:

```
Before : 1, 2
After: 2, 1
Before : 3, 5
After: 5, 3
```

PRIMITIVES AND METHOD PARAMETERS

- Examine the following code and its output:

```java
package oop00;
public class MethodParametersTest3 {
    public static void main(String[] args) {
         int[] dizi = { 1, 2, 3, 4, 5 };
         LowHighSwap.doIt( dizi );
         for( int j = 0; j < dizi.length; j++ )
             System.out.print( dizi[j] + " " );
    }
}
class LowHighSwap {
    static void doIt( int[] z ) {
         int temp = z[ z.length - 1 ];
         z[ z.length - 1 ] = z[ 0 ];
         z[ 0 ] = temp;
    }
}
/* Using static has nothing to do with the
 * "call-by-value-of-references" issue. */
```

- The output:

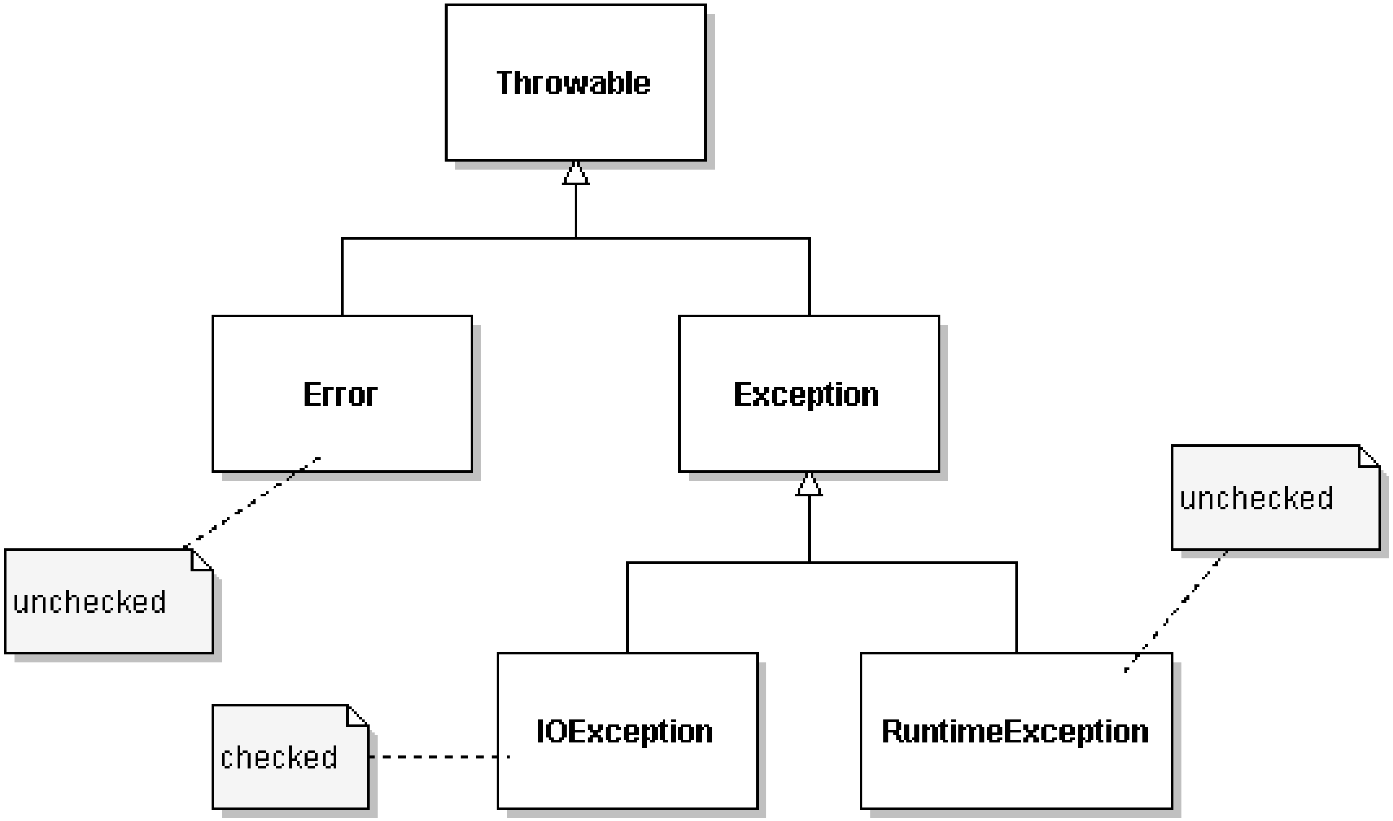
```
5 2 3 4 1
```

EXCEPTION HANDLING

- "If that guy has any way of making a mistake, he will"
  - Murphy's Law
- Some sources of error are:
  - Bugs in JVM
  - Wrong input by the user
  - Buggy code written by us
  - Acts of God
    - A lone and humble programmer cannot control:
      - every aspect of Internet traffic,
      - file access rights,
      - etc.
    - But we should be aware of them and deal with them!
- There are multiple ways of dealing with errors.
  - Boolean returns
  - Form components with error checking mechanisms
  - Exception handling.
- Exception handling is a form of error trapping.

EXCEPTION HANDLING

- Each exception is modeled by a class in Java.



EXCEPTION HANDLING

- java.lang.Error:
  - indicates serious problems that a reasonable application should not try to catch
    - Depletion of system resources, internal JVM bugs, etc.
    - java.lang.UnsupportedClassVersionError: Can happen when you move your code between different versions of Eclipse.
- java.lang.RuntimeException:
  - This is mostly caused by our buggy code
    - java.lang.NullPointerException: We have tried to use an uninitialized object
    - java.lang.IndexOutOfBoundsException: We have tried to access a non-existent member of an array.
    - etc.
- java.io.IOException:
  - Something went wrong during a file operation or a network operation.
  - These operations are always risky, so we must have an alternate plan in case of something goes wrong.
    - If having an alternate plan is a must, than the exception is determined as checked.

EXCEPTION HANDLING

- Handling checked exceptions is done by coding a try – catch block.

```
try {
    /* error-prone methods */;
}
catch( AnException e ) {
    /* Dealing with error */
}
```

- A programmer may opt to not handle a checked exception.
    - However, someone will eventually handle it!

```
aMethod(…) throws AnException {
    /* error-prone methods */
}
```

    - In this case, this someone is the one who calls that aMethod

EXCEPTION HANDLING

- It is possible to handle multiple exceptions as well:

```
try {
   /* error-prone methods */;
}
catch( AnException e ) {
   /* Dealing with error */
}
catch( AnotherException e ) {
   /* Dealing with error */
}
```

- About try blocks:
  - Each new try block introduces a runtime overhead
  - Therefore it's wiser to open one try block with multiple catch blocks

EXCEPTION HANDLING

- What should I do in a catch block?
  - Inform the user about the error with the e.printStackTrace( ) method.
  - Log this error
- If this is a very serious error, you may release some resources and make a "clean exit" in the finally block.
  - Scopes of the try block and the finally block are different. Therefore you cannot access the temporary variables/objects defined in the try block from the finally block. Plan your "clean exit" accordingly.
  - The finally block executes whether an exception is thrown or not.

```
try {
    /* error-prone methods */;
}
catch( AnException e ) {
    /* Dealing with error */
}
catch( AnotherException e ) {
    /* Dealing with error */
}
finally {
    /* make a clean exit */
}
```

EXCEPTION HANDLING

```
public class ExceptionExample01 {
    MyScreenRenderer graphics;
    MyCADfile myFile;
    //Other methods of this class are omitted
    public void parseMyCADfile( String fileName ) {
        try {
            graphics = new MyScreenRenderer();
            myFile = openFile( fileName );
            MyFigure figs[ ] = myFile.readFromFile( );
            drawFigures( figs );
            myFile.close();
        }
        catch( IOException e ) {
            System.out.println("An IO exception has occurred"+
                " while opening or reading from file:"+
                e.toString( ) );
            e.printStackTrace( );
            System.exit(1);
        }
        finally {
            graphics.releaseSources();
        }
    }
}
```

EXCEPTION HANDLING

- You can create your own Exception classes by :
  - inheriting from IOException if you want your exception to be a checked one,
  - inheriting from RuntimeException if you want an unchecked one.

```
public class MyFileFormatException extends IOException {
    public MyFileFormatException( ) {
        super( );
    } //was required in JDK versions older than 5
    public MyFileFormatException( String errorMessage ) {
        super( errorMessage );
        /* Ayrıca yapmak istediğiniz işler */
    }
}
```

EXCEPTION HANDLING

- Throwing an exception:
  - If something terrible may happen during your code, you can throw an exception

```
public class AProgram {
    public void processFile ( ) throws MyFileFormatException{
        some_statements();
        if( an_unexpected_situation )
                throw new MyFileFormatException("... happened");
    }
}
```

EXCEPTION HANDLING

- Let's wrap it up all by an example:

```
package oop01;
import java.io.IOException;
    /* Eğer RuntimeException'dan türetsek
     * tüm kodlarımız nasıl olacaktı inceleyiniz.
     */
    @SuppressWarnings("serial")
    public class ImpossibleInfo extends IOException {
       public ImpossibleInfo( String errorMessage ) {
             super(errorMessage);
       }
    }
}
```

EXCEPTION HANDLING

```
package oop01;
public class Person {
    private String name;
    private int age;

    public Person( String name ) { this.name = name; }
    public String getName( ) { return name; }
    public int getAge( ) { return age; }
    public String toString() {
        return getName() + " " + getAge( );
    }
    public void setAge( int age ) throws ImpossibleInfo {
        if( age < 0 || age > 150 )
            throw new ImpossibleInfo("Impossible age: "+age);
        this.age = age;
    }
}
```

EXCEPTION HANDLING

```
package oop01;
import java.util.*;
public class TestExceptions {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter person's name: ");
        String name = in.nextLine();
        Person insan = new Person(name);
        try {
                System.out.print("Enter age: ");
                int age = in.nextInt();
                insan.setAge(age);
                System.out.println(insan);
        }
        catch (ImpossibleInfo e) {
                e.printStackTrace();
        }
        finally {
                in.close();
        }
    }
}
```

- You can wrap the statement that can cause an exception and the remaining statements with the try block or you can put all statements into the try block.

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Generic programming means to write code that can be reused for objects of many different types.
  - Recent languages such as Java and C# support generics.
  - Generics are, at least on the surface, similar to templates in C++.
- The aim of this section is to make you familiar with the usage of generic classess.
  - We will try to reach this aim by teaching you how to use some of the generic classes that comes with the Java language.
  - We have chosen the examples from the basic data structures in the java.util library.
  - Therefore, an introduction to these data structures exists in the course notes.
- The aim of this section is not to:
  - Teach you how to write your own generic classes.
  - Teach you about data structures.

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

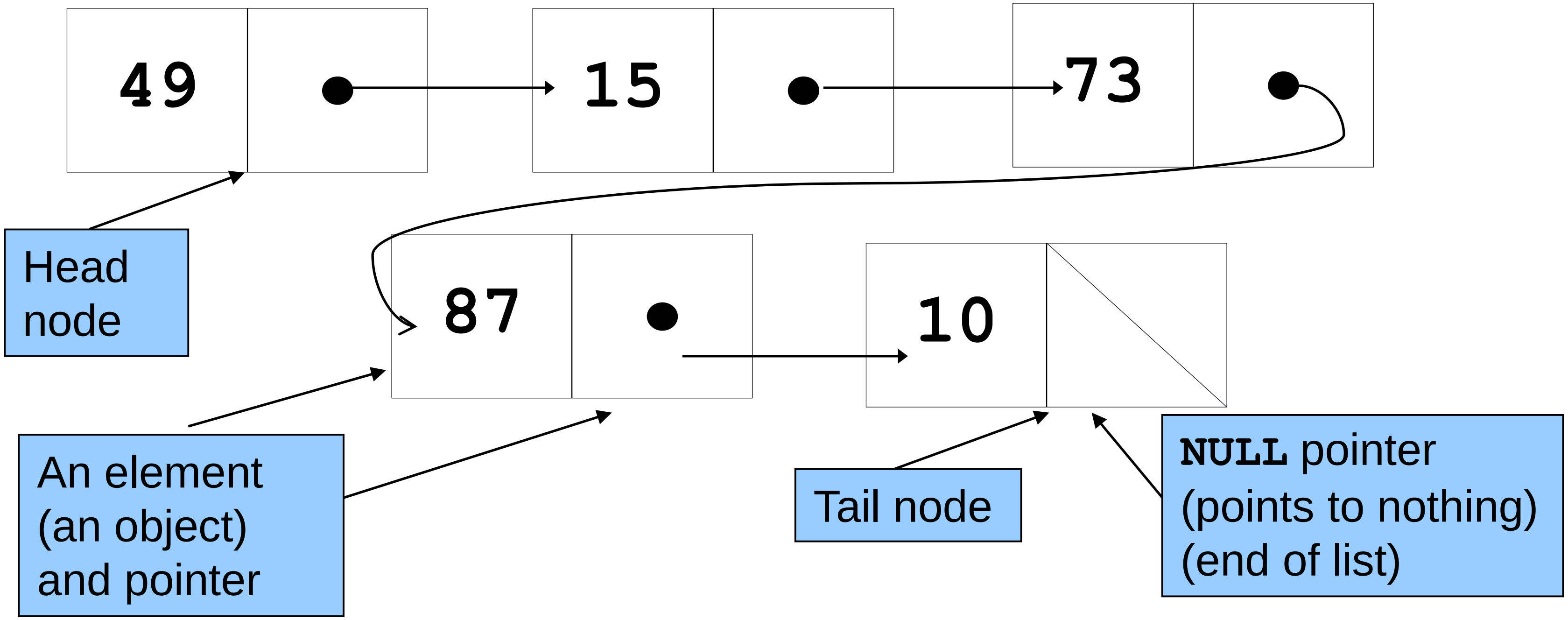
- Introduction to Data Structures:
  - A data structure is a scheme for organizing data in the memory of a computer
  - In a general sense, any data representation is a data structure.
    - Example: An integer.
  - More typically, a **data structure is** meant to be **an organization for a collection of data items**.
  - The way in which the data is organized affects the performance of a program for different tasks.
  - The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days!
  - Some of the more commonly used data structures include arrays, lists, stacks, queues, heaps, trees, and graphs.

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Introduction to linked lists:
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links (transparent to the programming user in Java)
  - The first (head) and last (tail) nodes of the list are accessed via an object reference
  - Traversing between the nodes is done by using using an iterator object obtained from the data structure
    - You may write your own traversal code according to the organization of the data structure.
  - Traversing an entire list is easier (thanks to the for-each loop).

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- An example linked list holding Integer objects:

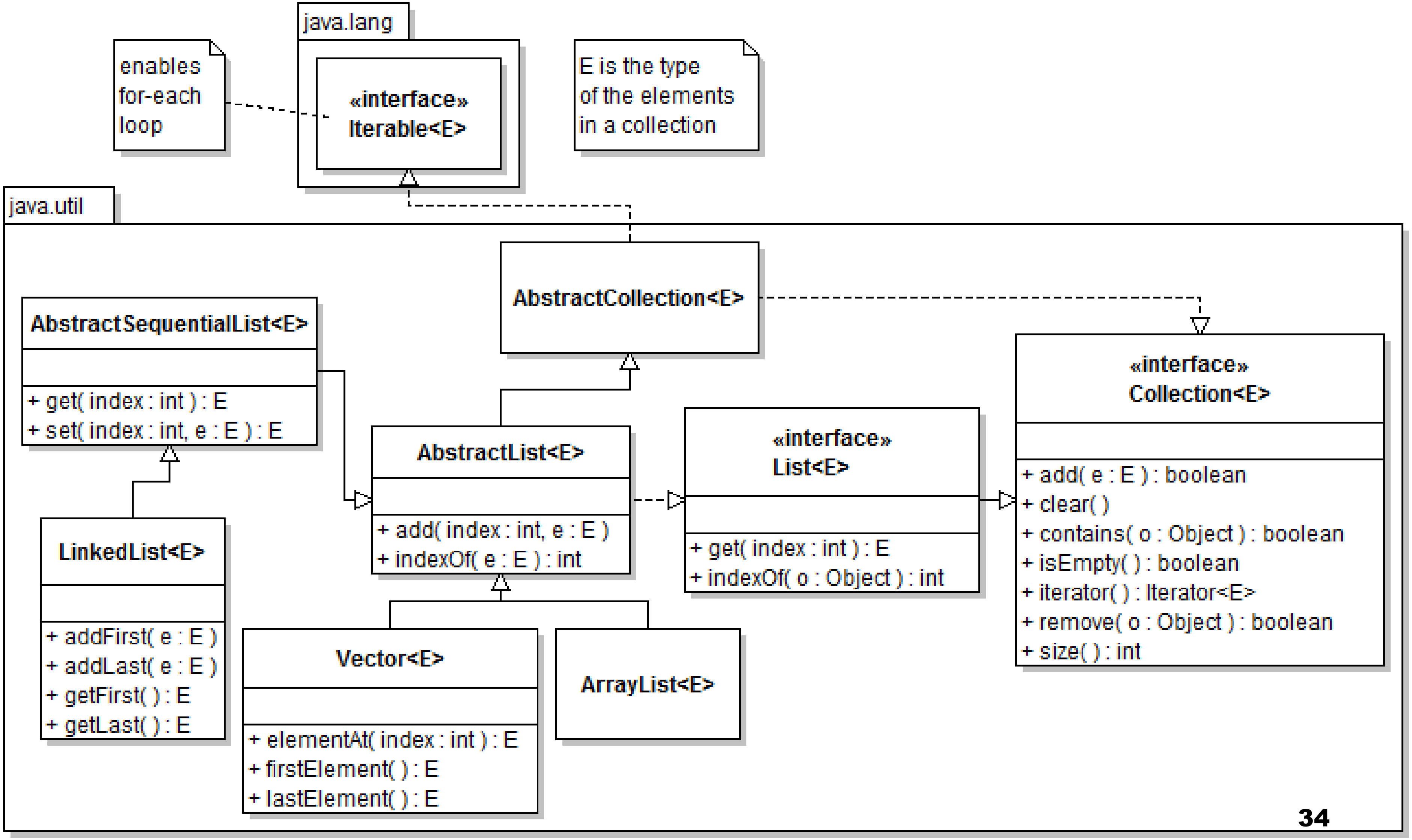

49
15
73
87
10
Head node
An element (an object) and pointer
Tail node
NULL pointer (points to nothing) (end of list)

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Advantages of linked lists over arrays:
  - Enlarging a list costs nothing!
  - Insertion and removal of elements to any position is faster.
  - Sorting algorithms work faster on linked lists.
- Advantage of arrays over linked lists:
  - Lists are traversed sequentially where any ith member of an array is directly accessible.
- Types of linked lists:
  - Single-linked list: Only traversed in one direction
  - Doubly-linked list: Allows traversals both forwards and backwards
- A list may also be circular.
  - Pointer in the last node points back to the first node

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Part of the inheritance tree of list structures in Java:



INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- An example list implementation in java: The java.util.ArrayList class
  - Based on arrays, single-linked, thread-unsafe.
  - Initialization:

```
ArrayList myList = new ArrayList();
```

- In such definition, the nodes are instances of Object.
  - This makes typecasting mandatory in order to use the node objects.
  - Luckily, support for 'Generic Programming' is introduced in JSE 5.0.
- **Generic programming** means to write code that can be reused for objects of many different types.
  - For example, you don’t want to program or use separate classes to collect String and File objects.
  - And you don’t have to!
    - The single class ArrayList collects objects of any class and constitutes an example of generic programming.
- Example: Creating an ArrayList instance which will contain Person instances

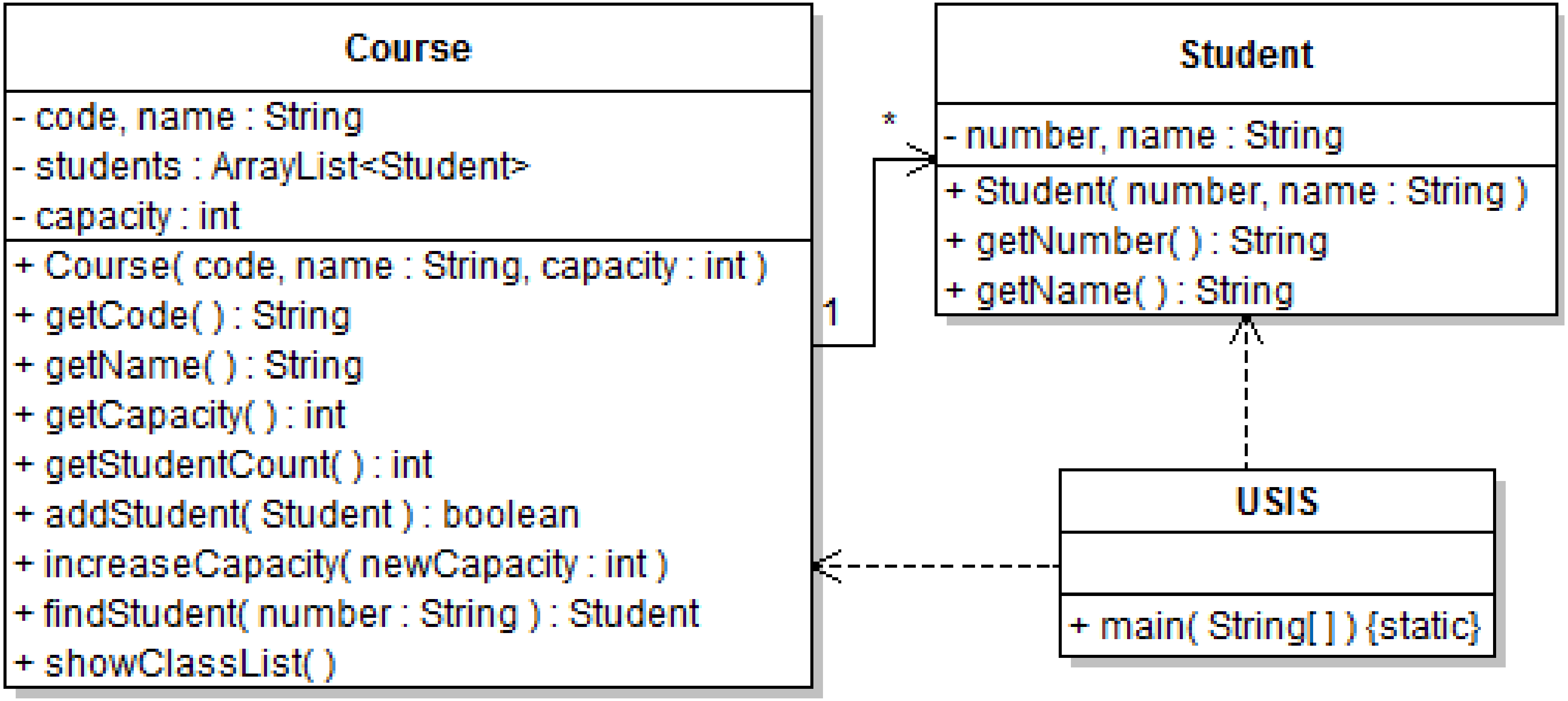
```
ArrayList<Person> liste = new ArrayList<Person>();
```

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Fundamental methods of the ArrayList class:
  - add(<T> object): Adds an element (an object of type T) to the end of the list.
  - <T> get(int i): Returns the ith element.
  - int size( ): Returns the number of elements in this list
  - Remember that the entire list can be easily traversed by using the for-each loop.
- A selection of the other methods of the ArrayList class:
  - ensureCapacity(int size): Increases the capacity of this ArrayList instance, if necessary.
  - trimToSize( ): Trims the capacity of this ArrayList instance to be the list's current size.
  - set(int i, <T> element): Replaces the element at the specified position in this list with the specified element.
  - remove(int i): Removes the ith element from this list
    - If the current size is less than i, an IndexOutOfBoundsException is throwed (unchecked).

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Let's implement a multiplicty association (1-*) by using ArrayList



Course

- code, name : String
- students : ArrayList<Student>
- capacity : int

+ Course(code, name : String, capacity : int)
+ getCode( ) : String
+ getName( ) : String
+ getCapacity( ) : int
+ getStudentCount( ) : int
+ addStudent(Student) : boolean
+ increaseCapacity(newCapacity : int)
+ findStudent(number : String) : Student
+ showClassList( )

Student

- number, name : String

+ Student(number, name : String)
+ getNumber( ) : String
+ getName( ) : String

USIS

+ main(String[ ]) {static}

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

```java
package oop02a;
import java.util.*;

public class Course {
    private String code; private String name; private int capacity;
    private ArrayList<Student> students;

    public Course(String code, String name, int capacity) {
        this.code = code; this.name = name; this.capacity = capacity;
        students = new ArrayList<Student>();
    }
    public String getCode() { return code; }
    public String getName() { return name; }
    public int getCapacity() { return capacity; }
    public int getStudentCount() {
        return students.size();
    }
    public boolean addStudent( Student aStudent ) {
        if( getStudentCount() == capacity ||
                        findStudent(aStudent.getNumber()) != null )
                return false;
        students.add(aStudent);
        return true;
    }
```

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

```
    public Student findStudent( String number ) {
        for( Student aStudent : students )
                if( aStudent.getNumber().compareTo(number) == 0 )
                        return aStudent;
        return null;
    }
    public void increaseCapacity( int newCapacity ) {
        if( newCapacity <= capacity )
                return;
        capacity = newCapacity;
    }
    public void showClassList( ) {
        System.out.println("Class List of "+code+" "+name);
        System.out.println("Student#  Name, Surname");
        System.out.println("--------  -----------------------------");
        for( Student aStudent : students )
                System.out.println(aStudent.getNumber()+
                " " + aStudent.getName());
    }
}
```

- Please compare this code with the ones you can write by using arrays and see how much cleaner your code has become (show from oop02x).
  - In this example, the member "capacity" exists only for business logic, not for array operations.

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

```
package oop02x;
public class Course {
    private String code; private String name; private int capacity, studentCount;
    private Student[ ] students;
    public Course(String code, String name, int capacity) {
         this.code = code; this.name = name; this.capacity = capacity;
         students = new Student[capacity]; studentCount = 0;
    }
    public String getCode() { return code; }
    public String getName() { return name; }
    public int getCapacity() { return capacity; }
    public int getStudentCount() { return studentCount; }
    public boolean addStudent( Student aStudent ) {
         if( studentCount == capacity || findStudent(aStudent.getNumber()) != null )
                  return false;
         students[studentCount] = aStudent;
         studentCount++;
         return true;
    }
    public Student findStudent( String number ) {
         for( int i = 0; i < studentCount; i++ )
                  if( students[i].getNumber().compareTo(number) == 0 )
                           return students[i];
         return null;
    }
```

- Continues on the next slide

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Continued from the next slide

```
    public void increaseCapacity( int newCapacity ) {
         if( newCapacity <= capacity )
                  return;
         Student[ ] geciciDizi = new Student[ newCapacity ];
         for( int i = 0; i < studentCount; i++ )
                  geciciDizi[i] = students[i];
         students = geciciDizi;
         capacity = newCapacity;
    }
    public void showClassList( ) {
         System.out.println("Class List of "+code+" "+name);
         System.out.println("Student#  Name, Surname");
         System.out.println("--------  ------------------------------");
         for( Student aStudent : students )
             if( aStudent != null ) //dizi gerçeklemesinde gerekli!
                  System.out.println(aStudent.getNumber()+" "+aStudent.getName());
    }
}
```

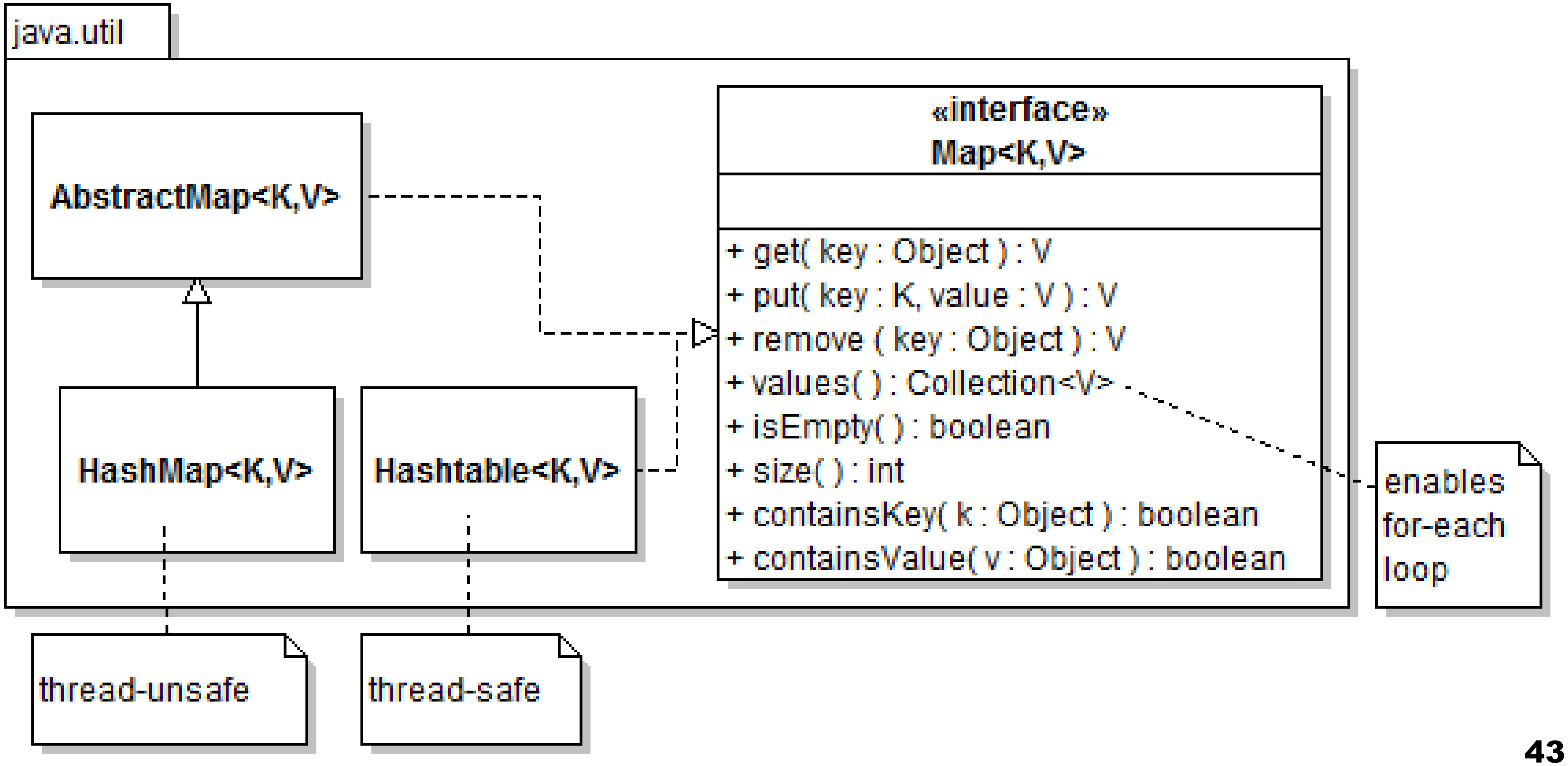
INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Now, let’s demonstrate and test our code:

```
public class USIS {
     public static void main(String[] args) {
         Course oop = new Course("0112562", "Obj. Or. Prog.", 3);
         Student student1 = new Student("09011034",“Student#1");
         if( !oop.addStudent(student1) )
         System.out.println("Problem #1");
         boolean result;
         result = oop.addStudent(student1);
         if( result == true )
              System.out.println("Problem #2");
         Student student2 = new Student("09011045",“Student#2");
         oop.addStudent(student2);
         Student student3 = new Student("09011046",“Student#3");
         oop.addStudent(student3);
         Student student4 = new Student("09011047",“Student#4");
         if( oop.addStudent(student4) )
              System.out.println("Problem #3");
         if( oop.findStudent("09011046") != student4 )
              System.out.println("Problem #4");
         if( oop.findStudent(student4.getNumber()) == null )
              System.out.println("Problem #5");
         System.out.println("End of test\n");
         oop.showClassList();
     }
}
```

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Introduction to map structures:
  - The map data structure lets you to easily reach an existing element according to its unique identifier
    - This operation is also much faster with map structures than with array and list structures
  - Element = value, unique identifier = key


java.util
«interface»
Map<K,V>
+ get(key : Object) : V
+ put(key : K, value : V) : V
+ remove (key : Object) : V
+ values( ) : Collection<V>
+ isEmpty( ) : boolean
+ size( ) : int
+ containsKey(k : Object) : boolean
+ containsValue(v : Object) : boolean
AbstractMap<K,V>
HashMap<K,V>
Hashtable<K,V>
enables for-each loop
thread-unsafe
thread-safe

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- A map implementation in Java: The java.util.HashMap class
  - java.util.HashMap<K,V>
    - K: Key, V: Value
- Fundamental methods of the HashMap class :
  - public V get(Object key);
    - Returns the value to which the specified key is mapped.
  - public V put(K key, V value);
    - Associates the specified value with the specified key in this map.
      - I suggest you to obtain the key from the value (by using the necessary get method to access its unique identifier).
      - If a value already exists in the data structure with the given key, the old value is deleted and returned, whereas the new value is inserted into the collection.
  - public Collection<V> values( );
    - Returns a list of all values stored in this table which can easily be traversed by using the for-each loop.
    - At this point, it is not necessary to know the specifics of the generic Collection interface.

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- Let's implement the previous example by using HashMap:

```
package oop02b;
import java.util.*;

public class Course {
    private String code; private String name; private int capacity;
    private HashMap<String,Student> students;

    public Course(String code, String name, int capacity) {
        this.code = code; this.name = name; this.capacity = capacity;
        students = new HashMap<String,Student>();
    }
    public String getCode() { return code; }
    public String getName() { return name; }
    public int getCapacity() { return capacity; }
    public int getStudentCount() {
        return students.size();
    }
    public boolean addStudent( Student aStudent ) {
        if( getStudentCount() == capacity ||
                      findStudent(aStudent.getNumber()) != null )
               return false;
        students.put(aStudent.getNumber(), aStudent);
        return true;
    }
```

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

```
    public Student findStudent( String number ) {
        return students.get(number);
    }
    public void increaseCapacity( int newCapacity ) {
        if( newCapacity <= capacity )
                return;
        capacity = newCapacity;
    }
    public void showClassList( ) {
        System.out.println("Class List of "+code+" "+name);
        System.out.println("Student#  Name, Surname");
        System.out.println("--------  -----------------------------");
        for( Student aStudent : students.values() )
                System.out.println(aStudent.getNumber()+
                " " + aStudent.getName());
    }
}
```

- Please compare this code which uses maps with the previous one which uses lists and notice the conveniences for the programmer.

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

TESTING

- We wrote some code, but we didn't test it for bugs. Let's do it:

```
public class USIS {
    public static void main(String[] args) {
        Course oop = new Course("0112562", "Obj. Or. Prog.", 3);
        Student yasar = new Student("09011034","Yaşar Nuri Öztürk");
        if( !oop.addStudent(yasar) )
                System.out.println("Problem #1");
        if( oop.addStudent(yasar) )
                System.out.println("Problem #2");
        Student yunus = new Student("09011045","Yunus Emre Selçuk");
        oop.addStudent(yunus);
        Student fatih = new Student("09011046","Fatih Çıtlak");
        oop.addStudent(fatih);
        Student cemalnur = new Student("09011047","Cemalnur Sargut");
        if( oop.addStudent(cemalnur) )
                System.out.println("Problem #3");
        if( oop.findStudent("09011046") != fatih )
                System.out.println("Problem #4");
        System.out.println("End of test");
    }
}
```

Why isn't there a package name?

How can we document this new class?

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

SUMMARY OF FUNDAMENTAL DATA STRUCTURE IMPLEMENTATIONS:

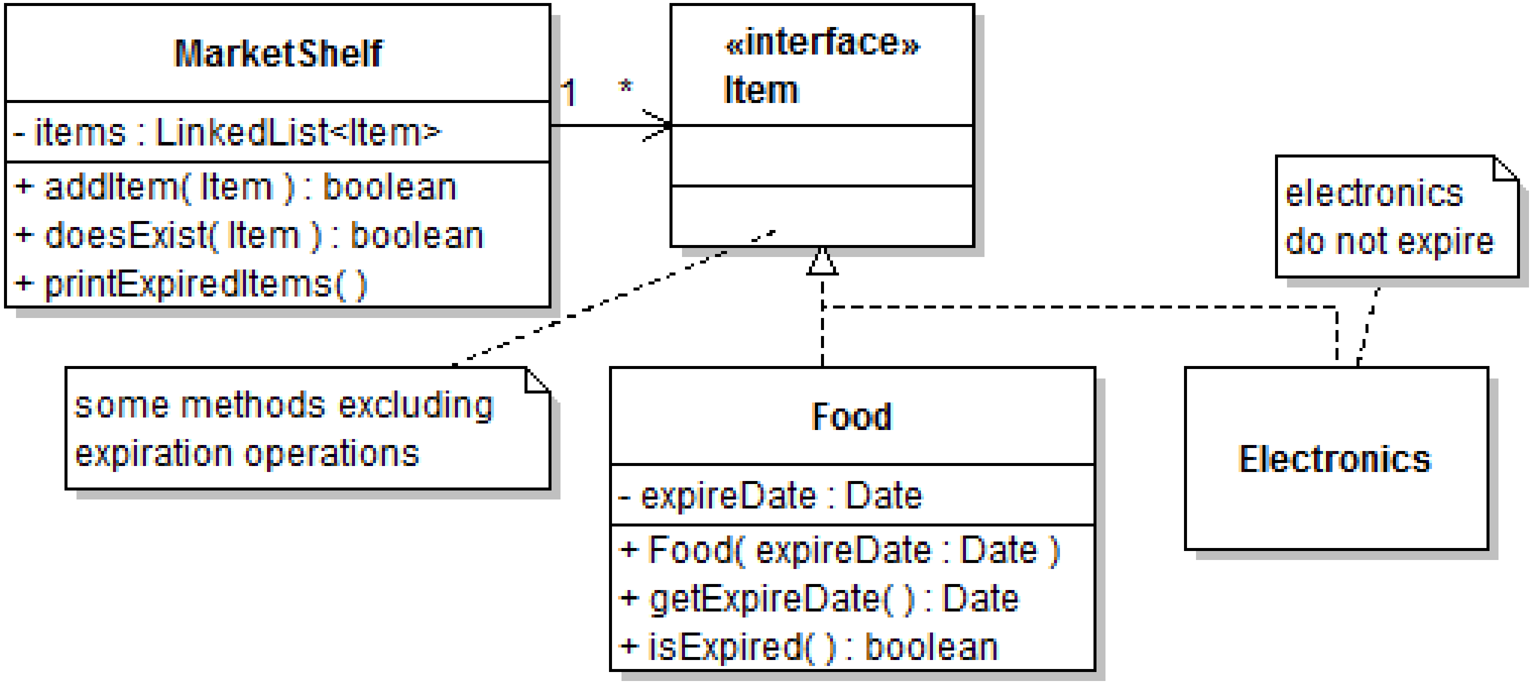
- java.util.LinkedList<E> implements List<E>
  - Faster insertions and deletions
  - Slower random access
  - Doubly-linked (Can be traversed backwards by obtaining a ListIterator instance [not to be covered?]).
- java.util.ArrayList<E> implements List<E>
  - Slower insertions and deletions
  - Faster random access
- java.util.Vector<E> implements List<E>
  - Similar to ArrayList
  - synchronized
    - Suitable for multi-threaded use, slower in single-threaded use
- java.util.HashMap<K,V> implements Map<K,V>
  - Used for fast searches by a key (indexed)
- java.util.Hashtable<K,V> implements Map<K,V>
  - Similar to HashMap but synchronized
    - Suitable for multi-threaded use, slower in single-threaded use
  - Attention: Lowercase t in class name Hashtable

TYPECASTING

- Remember the following rule of inheritance:
  - An instance of a sub class can be used wherever an instance of its super class is expected.
- This is a type-safe operation and it is done automatically.
  - We can convert a specific object to a more general one without loosing any information.
  - Conversion in the opposite direction is risky, therefore it is done manually.
- The Java terminology uses the word "type casting" for converting the type of an object.
- You can make a manual cast from one type to another, according to the following rules:
  - From the interface to the class of the object
  - From the super class to the sub class
- However, this is an unsafe operation and therefore you need to make a check beforehand

TYPECASTING

- Example:



TYPECASTING

- Coding class MarketShelf:

```
import java.util.*;
public class MarketShelf {
   private LinkedList<Item> items;
   public MarketShelf() {
       items = new LinkedList<Item>();
   }
   public boolean doesExist( Item anItem ) {
       for( Item item : items )
       if( item == anItem )
             return true;
       return false;
   }
   public boolean addItem( Item anItem ) {
       if( doesExist(anItem) )
             return false;
       items.add(anItem);
       return true;
   }
```

TYPECASTING

- Coding class MarketShelf:

```
public void printExpiredItems( ) {
    boolean hasExpiredItem = false;
    System.out.println("Expired item(s): ");
    for( Item item : items ) {
        if( item instanceof Food )
            if( ((Food)item).isExpired() ) {
                hasExpiredItem = true;
                System.out.println(item);
            }
    }
    if( hasExpiredItem == false )
            System.out.println("All items are fresh!");
}
```

- <u>Checking for type compliance and typecasting</u>

TYPECASTING

- Coding class MarketShelf:

```
   public static void main( String[] args ) {
      MarketShelf shelf = new MarketShelf();
      Calendar cal = Calendar.getInstance();
      cal.add(Calendar.DAY_OF_MONTH,1);
      Date future = cal.getTime();
      cal.set(Calendar.YEAR, 2010);
      cal.set(Calendar.MONTH, 0); //0: January
      cal.set(Calendar.DATE, 12);
      Date past = cal.getTime();
      shelf.addItem( new Food(past) );
      shelf.addItem( new Food(future) );
      shelf.addItem( new Electronics() );
      shelf.checkForExpiration();
   }
}
```

- Using java.util.Date and Calendar classes

TYPECASTING

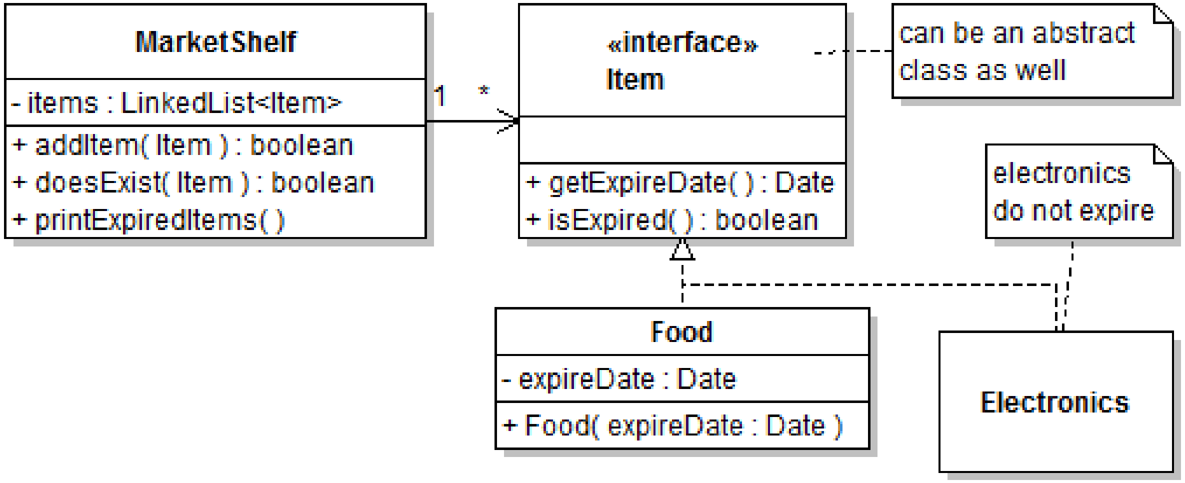
- Coding class Food:

```
package oop03b;
import java.util.Date;
public class Food implements Item {
    private Date expireDate;

    public Food(Date expireDate) {
        this.expireDate = expireDate;
    }
    public Date getExpireDate() { return expireDate; }
    public boolean isExpired( ) {
        Date today = new Date();
        if( expireDate.before(today) )
                return true;
        else return false;
    }
    public String toString() {
        return "A food expiring at " + expireDate;
    }
}
```

- Using java.util.Date class continues

TYPECASTING

- Critique of typecasting:
  - Typecasting is a "necessary evil". Use it sparingly.
    - Back in the days where generic classes were not available in Java, we had to make typecasting frequently.
    - Nowadays, we need typecasting only when we make deserialization (topic of the next lecture).
  - We can always make designs without typecasting.
    - Let's modify our design:



MarketShelf
- items : LinkedList<Item>
+ addItem(Item) : boolean
+ doesExist(Item) : boolean
+ printExpiredItems( )
1
*
«interface»
Item
+ getExpireDate( ) : Date
+ isExpired( ) : boolean
can be an abstract class as well
electronics do not expire
Food
- expireDate : Date
+ Food(expireDate : Date)
Electronics

  - In other cases, we can make good use of abstract classes and polymorphism.
    - In those cases, we are advised to avoid any relationship with the subclasses.

WORKING WITH FILES

RELATED EXCEPTIONS

- java.io.IOException: Represents I/O exceptions in general.
- java.io.EOFException extends IOException: Indicates that the end of file or stream has been reached unexpectedly.
- java.io.FileNotFoundException extends IOException: Indicates that the requested file cannot be found in the given path.
- java.lang.SecurityException extends java.lang.RuntimeException: Indicates that the requested operation cannot be executed due to security constraints.

GENERAL INFORMATION ABOUT FILE OPERATIONS

- File operations are separated into two main groups in Java:
  - File management: Opearations such as creating, renaming, deleting files and folders.
  - I/O operations.
- I/O operations are not only done with files but also with different sources such as TPC sockets, web pages, console, etc. Therefore I/O operations:
  - have been separated from file operations
  - coded in the same way for all these different sources.
- This approach is in harmony with the nature of object oriented paradigm. However, the complexity has been increased as a side effect.

WORKING WITH FILES

FILE MANAGEMENT

- Coded by using the java.io.File class which represents both the files and the folders in the hard drive.
- Creating a File object does not mean to create an actual file or folder.
- Creating a File object :
  - Done by using the File(String fileName) constructor.
  - fileName should contain both the path and the name of the file/folder.
    - Full path vs. relative path.
      - Using full path degrades portability
      - Relativity is tricky as well: IDEs may keep source and class files in different folders.
    - Path separator:
      - Windows uses \ (should be denoted as \\ in Strings), Unix uses /.
      - What about portability?
        - public static String File.separator
        - public static char File.separatorChar
  - File(String path, String name) and File(File path, String name) constructors:
    - Represents a file/folder with the given name in the folder given by the path parameter.

WORKING WITH FILES

FILE MANAGEMENT

- Some methods of the class java.io.File:
  - boolean exists( ); tells whether the file exists or not.
  - boolean isFile( ); returns true if this File object represents a file, false otherwise, i.e. this object represents a folder.
  - File getParentFile( ); Returns the directory where this file/folder resides.
  - String getCanonicalPath( ) throws IOException; Returns the full path of the file/folder, including the file name.
  - boolean canRead( ); Can this application read form this file?
  - boolean canWrite( ); Can this application write to this file?
  - boolean createNewFile( ); Actually creates the file.
  - boolean mkdir( ); Actually creates the folder.
  - boolean mkdirs( ); Actually creates the folder with all necessary parent folders
  - boolean renameTo(File newName); Renames the file.
  - boolean delete( ); Deletes the file.
- boolean returns: True if the operation is successful.
- You do not have to memorize all those methods.

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Any I/O source is represented as stream in Java
  - Files, memory, command prompt, network, etc.
- Binary vs. Text format:
  - Binary I/O is fast and efficient, but it is not easily readable by humans.
  - Text I/O is the opposite.
- Random vs. Sequential access:
  - Sequential access: All records are accessed from the beginning to the end
  - Random access: A particular record can be accessed directly.
  - Disk files are random access, but streams of data from a network are not.
- Java chains streams together for different working styles.
- We will study a mechanism which allows makes it possible to write any object to a stream and read it again later.
  - This process is called serialization in the Java terminology.

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Serialization – Output operations:
  - We will write entire objects to a file on disk.
  - The classes of objects to be serialized should implement the java.io.Serializable interface.
  - You do not need to do anything else as the java.io.Serializable interface does not have any methods.
  - ObjectOutputStream and FileOutputStream objects are chained together for serialization.
  - Multiple objects can and should be sent to the same stream.

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Example record: the class Arkadas

```
public class Arkadas implements java.io.Serializable {
   private static final long serialVersionUID = 1L;
   private String isim, telefon, ePosta;
   public Arkadas( String name ) { this.isim = name; }
   public String getIsim( ) { return isim; }
   public String getTelefon( ) { return telefon; }
   public void setTelefon( String telefon ) {
      this.telefon = telefon;    }
   public String getEPosta( ) { return ePosta; }
   public void setEPosta( String posta ) { ePosta = posta; }
   public String toString( ) {
      return isim + " - " + telefon + " - " + ePosta;
   }
}
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- About the lines beginning with @ :
  - These are special commands called “annotations”.
  - They work at the “meta” level, i.e. they contain “information about information”.
  - They give information to the IDE, compiler, another programme, etc. about this program.
  - We have used the annotation mechanism to remove the warnings.
  - In fact, warnings must be taken into consideration. In the previous examples, we have disabled these warnings with annotations.
  - In the example above, we didn’t use annotation as the warning is directly related with our current subject.
- About “marking interfaces”:
  - The java.io.Serializable interface does not include any methods to be implemented. This interface is used only for marking/highlighting the classes where its instances are to be serialized.

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- About the serialVersionUID member:
  - private static final long serialVersionUID = 1L;
  - We can give a particular version instead of 1, or we can have the IDE to generate a unique identifier automatically.
  - If we do not code this member, we can hide the related warning with the @SuppressWarnings("serial") command.
  - What does this member mean?
    - There will be applications which save and load objects from different sources.
    - In time, the source code of the classes of these objects may change, as well as the source code of the aforementioned applications.
    - Different versions of all those classes can exist together. In order to avoid incompatibilities, we need a versioning mechanism.
    - This mechanism is implemented by giving a different (and possibly increasing) serial number to classes and by checking this serial in the applications.

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- An application for writing the objects to a file (serialization/output):

```
import java.util.*;
import java.io.*;
public class ArkadasOlustur {
    public static void main(String[] args) {
        Integer arkadasSayisi;
        Arkadas[ ] arkadaslar;
        Scanner giris = new Scanner( System.in );
        System.out.println("Bu program arkadaşlarınızın iletişim " +
                         " bilgilerini diskteki bir dosyaya kaydeder.");
        System.out.print("Kaç arkadaşınızın bilgisini gireceksiniz? ");
        arkadasSayisi = giris.nextInt( );
        giris.nextLine( );
        arkadaslar = new Arkadas[arkadasSayisi];
        for( int i = 0; i < arkadasSayisi; i++ ) {
                System.out.print((i+1)+". arkadaşınızın ismi nedir? ");
                arkadaslar[i] = new Arkadas( giris.nextLine() );
                System.out.print("Bu arkadaşınızın telefonu nedir? ");
                arkadaslar[i].setTelefon( giris.nextLine() );
                System.out.print("Bu arkadaşınızın e-posta adresi nedir? ");
                arkadaslar[i].setEPosta( giris.nextLine() );
        }
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Serialization example (cont’d):

```
       try {
               String dosyaAdi = "arkadaslar.dat";
               ObjectOutputStream yazici = new ObjectOutputStream(
                              new FileOutputStream( dosyaAdi )  );
               yazici.writeObject( arkadasSayisi );
               for( Arkadas arkadas : arkadaslar )
                      yazici.writeObject( arkadas );
               yazici.close();
               System.out.println("Girilen bilgiler " + dosyaAdi +
                      " adlı dosyaya başarıyla kaydedildi.");
       }
       catch( IOException e ) {
               System.out.println("Dosyaya kayıt işlemi sırasında"+
                      " bir hata oluştu.");
               e.printStackTrace();
       }
    }
}
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Deserialization – Input operations:
    - We will read entire objects form a file on disk.
    - ObjectInputStream and FileInputStream objects are chained together for deserialization.
    - Typecasting is required as the objects read from a stream comes as instances of the class Object.
    - If these objects are to be stored in an array, we need to know how many objects there will be.
        - In the data structures that may grow dynamically, we are not faced with this inconvenience.

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- An application for reading the objects from a file (deserialization/input):

```
import java.io.*;

public class ArkadasGoster {
  public static void main( String[] args ) {
    String dosyaAdi = "arkadaslar.dat";
    try {
        ObjectInputStream okuyucu = new ObjectInputStream(
                       new FileInputStream( dosyaAdi ) );
        Integer kayitSayisi = (Integer)okuyucu.readObject();
        for( int i = 0; i < kayitSayisi; i++ ) {
               Arkadas arkadas = (Arkadas) okuyucu.readObject();
               System.out.println(arkadas);
        }
        okuyucu.close();
    }
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Deserialization example (cont’d):

```
    catch( IOException e ) {
        System.out.println("Dosya okuma işlemleri sırasında" +
                              " bir hata oluştu.");
        e.printStackTrace();
    }
    catch( ClassNotFoundException e ) {
        System.out.println("Okunan kayıtları işlerken " +
                       "bir hata oluştu.");
        e.printStackTrace();
    }
  }

}
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- More on object streams:
  - There is no safe and efficient way to determine whether the end of a stream has been reached. Therefore we couldn't use a while loop such as:

```
try {
    ObjectInputStream okuyucu = new ObjectInputStream(
                   new FileInputStream( dosyaAdi ) );
    Arkadas arkadas = (Arkadas) okuyucu.readObject();
    while okuyucu.hasNext() {
           System.out.println(arkadas);
           arkadas = (Arkadas) okuyucu.readObject();
    }
    okuyucu.close();
}
```

Does not work! Removed in JDK8.

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- There is a method, **`int ObjectInputStream.available( ),`** but this is somewhat buggy
  - http://www.coderanch.com/t/378141/java/java/EOF-ObjectInputStream
- Moreover, readObject() doesn't return null at EOF
  - http://stackoverflow.com/questions/2626163/java-fileinputstream-objectinputstream-reaches-end-of-file-eof
- You can code a solution by letting the exception to happen, and terminate the loop in the catch block.
  - However, exception handling is not invented for altering the program flow.
- A better alternative to writing the data object count beforehand is to use only one container object which stores references all the data objects.
  - This container object will be a **data structure,** such as a list or a map.
    - However, the objects in the container must implement the **`java.io.Serializable`** interface.
    - Will be shown in the next slide.
  - If there is a relation A→B, both A and B must implement the **`java.io.Serializable`** interface.

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Serializing data structures:

```
package oop04b;
import java.util.*;
import java.io.*;
@SuppressWarnings("resource")
public class ArkadasOlustur {
  public static void main(String[] args) {
    LinkedList<Arkadas> arkadaslar = new LinkedList<Arkadas>();
    Scanner giris = new Scanner( System.in );
    System.out.println("Bu program arkadaşlarınızın iletişim " +
        " bilgilerini diskteki bir dosyaya kaydeder.");
    System.out.print("Kaç arkadaşınızın bilgisini gireceksiniz? ");
    int arkadasSayisi = giris.nextInt( );
    giris.nextLine( );
    for( int i = 0; i < arkadasSayisi; i++ ) {
        System.out.print((i+1)+". arkadaşınızın ismi nedir? ");
        Arkadas arkadas = new Arkadas( giris.nextLine() );
        System.out.print("Bu arkadaşınızın telefonu nedir? ");
        arkadas.setTelefon( giris.nextLine() );
        System.out.print("Bu arkadaşınızın e-posta adresi nedir? ");
        arkadas.setEPosta( giris.nextLine() );
        arkadaslar.add(arkadas);
    }
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Serializing data structures (cont'd):

```
    try {
        String dosyaAdi = "arkadaslarAlt.dat";
        ObjectOutputStream yazici = new ObjectOutputStream(
                 new FileOutputStream( dosyaAdi )  );
        yazici.writeObject( arkadaslar );
        yazici.close();
        System.out.println("Girilen bilgiler " + dosyaAdi +
                 " adlı dosyaya başarıyla kaydedildi.");

    }
    catch( IOException e ) {
        System.out.println("Dosyaya kayıt işlemi sırasında"+
                 " bir hata oluştu.");
        e.printStackTrace();
    }
  }
}
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Deserializing data structures:

```
package oop04b;
import java.io.*;
import java.util.*;
public class ArkadasGoster {
  public static void main( String[] args ) {
    String dosyaAdi = "arkadaslarAlt.dat";
    try {
         ObjectInputStream okuyucu = new ObjectInputStream(
                 new FileInputStream( dosyaAdi ) );
         @SuppressWarnings("unchecked")
         LinkedList<Arkadas> arkadaslar =
                 (LinkedList<Arkadas>)okuyucu.readObject();
         for( Arkadas arkadas : arkadaslar ) {
                 System.out.println(arkadas);
         }
         okuyucu.close();
    }
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Deserializing data structures (cont'd.):

```
    catch( IOException e ) {
        System.out.println("Dosya okuma işlemleri sırasında" +
                 " bir hata oluştu.");
        e.printStackTrace();
    }
    catch( ClassNotFoundException e ) {
        System.out.println("Okunan kayıtları işlerken " +
                 "bir hata oluştu.");
        e.printStackTrace();
    }
  }
}
```

- What about working with text files or working in other modes?
  - Refer to Vol.II of Core Java 8th ed. or any other book of your choice.
  - Hint: PrintWriter and InputStreamReader streams are available for text output and input.

ENUM CLASSES

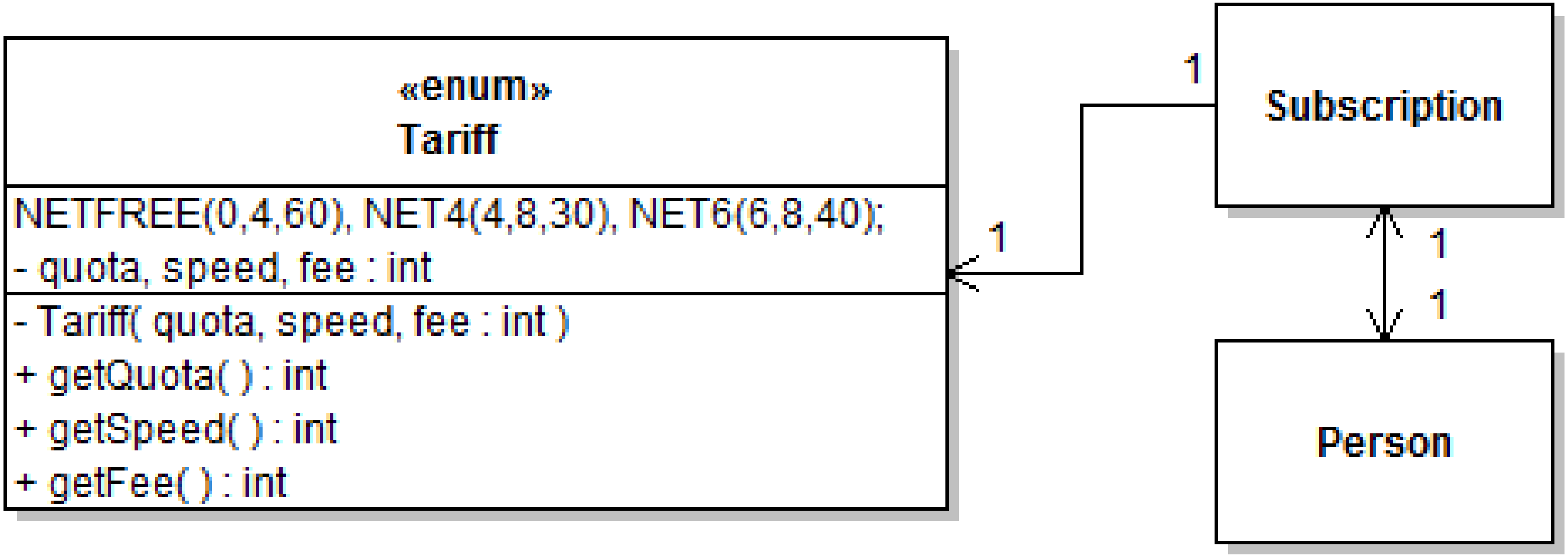
- The primitive enum we have learned is in fact a class definition.
- Each member of an enum is an instance of that class.
- There cannot be any other members of an enum except the ones that are already defined.
- An enum type can member fields, methods and constructors as any other regular classes.
- The constructor of an enum class must be private.
- Example:

This must be the first line of code!

```java
package oop05a;
public enum Tariff {
    NETFREE(0,4,60), NET4(4,8,30), NET6(6,8,40);
    private int quota, speed, fee;
    private Tariff( int quota, int speed, int fee ) {
        this.quota = quota; this.speed = speed; this.fee = fee;
    }
    public int getQuota() { return quota; }
    public int getSpeed() { return speed; }
    public int getFee() { return fee; }
}
```

ENUM CLASSES

- Example:



«enum»
Tariff

NETFREE(0,4,60), NET4(4,8,30), NET6(6,8,40);
- quota, speed, fee : int

- Tariff(quota, speed, fee : int)
+ getQuota( ) : int
+ getSpeed( ) : int
+ getFee( ) : int

Subscription

Person

ENUM CLASSES

- Creating and using an enum object:

```
public class Test {

   public static void main(String[] args) {
      Tariff tariff4 = Tariff.NET4;
      Person yunus = new Person("Yunus Emre");
      yunus.subscribeTo(tariff4);
      Person berkin = new Person("Berkin Gülay");
      berkin.subscribeTo(Tariff.NETFREE);
      System.out.println(yunus);
      System.out.println(berkin);
   }

}
```

INNER CLASSES

- You can code a class **<u>within</u>** a class.
  - An ***inner class*** is coded within an ***outer class***.
- An inner class can:
  - Access all members of the outer class, including the private ones.
  - Be hidden from other classes of the same package, if defined as private.
  - It is frequently used in form of ***anonymous inner classes*** in GUI programming.
    - Anonymous = without a name!
- You cannot:
  - define a static method in a an inner class.
- An example: Person and Employee classes

INNER CLASSES

```
package oop05b;
public class Person {
    private String name;
    public Person( String name ) { this.name = name; }
    @SuppressWarnings("unused")
    private class Employee { //begin inner class
        private int salary;
        public Employee( int salary ) { this.salary = salary; }
        public int getSalary() { return salary; }
        public void setSalary(int salary) { this.salary = salary; }
        public String toString( ) { return name + " " + salary; }
    } //end inner class
    public static void main( String[] args ) {
        Employee[] staff = new Employee[3];
        Person kisi;
        kisi = new Person("Osman Pamukoğlu");
        staff[0] = kisi.new Employee( 10000 );
        kisi = new Person("Nihat Genç");
        staff[1] = kisi.new Employee( 7500 );
        kisi = new Person("Barış Müstecaplıoğlu");
        staff[2] = kisi.new Employee( 6000 );
        for( Employee eleman: staff )
                System.out.println( eleman );
    }
}
```

INNER CLASSES

- Previous example is a demonstration of how to:
  - define an inner class
  - access the outer object from the inner object
- The inner class in the previous example is private.
  - Therefore, it is hidden from all classes, including the ones within the same package.
  - Which means, the Person.main method cannot be moved to any other class.
- The next example will show how to access a public inner class from any other class.

INNER CLASSES

```
package oop05c;

public class Person {
    private String name;
    public Person( String name ) { this.name = name; }

    public class Employee {
        private int salary;
        public Employee( int salary ) { this.salary = salary; }
        public int getSalary() { return salary; }
        public void setSalary(int salary) { this.salary = salary; }
        public String toString( ) { return name + " " + salary; }
    }
}
```

INNER CLASSES

```
package oop05c;

//this import is absolutely necessary
import oop05c.Person.Employee;

public class TestInnerClassDirectly {
    public static void main( String[] args ) {
        Employee[] staff = new Employee[3];
        Person kisi;
        kisi = new Person("Osman Pamukoğlu");
        staff[0] = kisi.new Employee( 10000 );
        kisi = new Person("Nihat Genç");
        staff[1] = kisi.new Employee( 7500 );
        kisi = new Person("Barış Müstecaplıoğlu");
        staff[2] = kisi.new Employee( 6000 );
        for( Employee eleman: staff )
                System.out.println( eleman );
    }
}
```

- PS: Instead of the import statement, you can write Person.Employee wherever necessary

INNER CLASSES

- This example shows how to access a private inner class from any other class:
  - By using a public method of the outer class
  - Meanwhile, we have to access the inner object from the outer object

```
package oop05d;
public class Person {
    private String name;
    private Employee employee;
    public Person(String name) { this.name = name; }
    public void enlist( int salary ) {
        employee = new Employee( salary ); }
    public String toString( ) {
        String mesaj = name;
        if( employee != null )
               mesaj += " " + employee.getSalary( );
        return mesaj;
    }
    @SuppressWarnings("unused")
    private class Employee {
        private int salary;
        public Employee( int salary ) { this.salary = salary; }
        public int getSalary() { return salary; }
        public void setSalary(int salary) { this.salary = salary; }
    }
}
```

INNER CLASSES

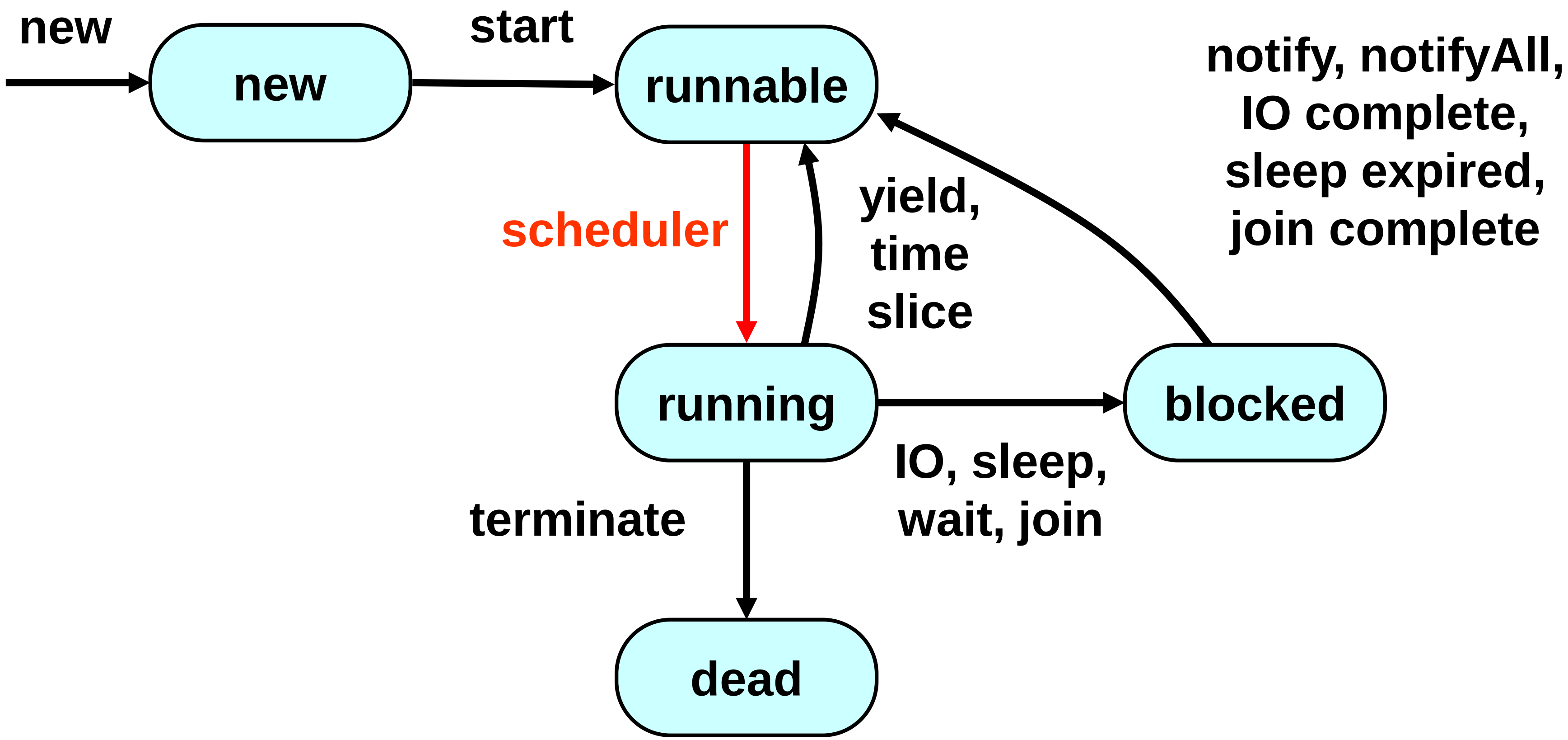
```
package oop05d;
public class TestInnerClassViaOuterClass {
    public static void main(String[] args) {
        Person[] staff = new Person[3];
        staff[0] = new Person( "Polat Alemdar" );
        staff[0].enlist( 10000 );
        staff[1] = new Person( "Memati Baş" );
        staff[1].enlist( 7000 );
        staff[2] = new Person( "Abdülhey Çoban" );
        staff[2].enlist( 5000 );
        for( Person insan: staff )
                System.out.println( insan );
    }
}
```

INTRODUCTION TO MULTITHREADING

- Multitasking, multiple processes and multithreading:
    - Multitasking is the ability to have more than one program working at the same time.
    - Nowadays, you are likely to have a computer with its CPU having multiple cores.
    - Each core can execute one or more tasks, i.e. processes, depending on the CPU architecture.
    - A process can sometimes be divided into threads that may run in parallel, i.e. concurrently running sub-processes.
        - If there are enough hardware resources, i.e. cores, the time it takes to complete a process will drop significantly.
        - However, this increase in the performance will not be in the order of the available cores.
            - The concurrently running threads will sooner or later need to synchonize with each other.
            - Moreover, creating a process or a thread takes some execution time as well.
- I have done significant simplifications while giving you this introduction!

INTRODUCTION TO MULTITHREADING

- A state diagram showing the possible states of a thread and transitions between those states:



INTRODUCTION TO MULTITHREADING

- How should a thread wait?
  - If a thread is unable to continue its task because of an obstacle, that thread should wait until the obstacle has been removed.
    - Obstacle: The needed information has not arrived from: the network, another thread, the user, etc.
  - You should not do “**busy waiting**”, i.e. executing dummy instructions such as running empty loops for 10.000 times.
  - Instead, you should put that thread into the blocked state by using the sleep command.
  - A sleeping thread, unlike a busy waiting one, does not consume system resources.
  - A sleeping thread is at risk of becoming unable to awake.
    - You must catch the java.lang.InterruptedException, which is a checked exception.

INTRODUCTION TO MULTITHREADING

- Procedure for running a task in a separate thread:
  1. Place the code for the task into the run method of a class that implements the Runnable interface.
  2. Create an object of your class
  3. Create a Thread object from the Runnable
  4. Start the thread by using Thread.start method (do not call the run method directly)
- Do not code your own threads by inheriting from the Thread class.
  - Otherwise you will lay your only inheritance right to waste.
- Let’s make a demonstration with a nonsense application about people watching a match:
  - Each person will shout for the team they support when he or she becomes excited.
  - There is a possibility for each person to become excited in 0-1000 ms.
  - Each person become exhausted after shouting 10 times.

INTRODUCTION TO MULTITHREADING

```
package oop06a;
import java.util.Random;

public class SoccerFan implements Runnable {
    public final static int STEPS = 10;
    public final static int DELAY = 1000;
    private String teamName, shoutPhrase;

    public SoccerFan( String teamName, String shoutPhrase ) {
        this.teamName = teamName;
        this.shoutPhrase = shoutPhrase;
    }

    public void run() {
        Random generator = new Random();
        try {
            for( int i = 0; i < STEPS; i++ ) {
                System.out.println( teamName + " " + shoutPhrase );
                Thread.sleep( generator.nextInt(DELAY) );
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

1. Place the code for the task into the run method of a class that implements the Runnable interface.

INTRODUCTION TO MULTITHREADING

3. Create a Thread object from the Runnable

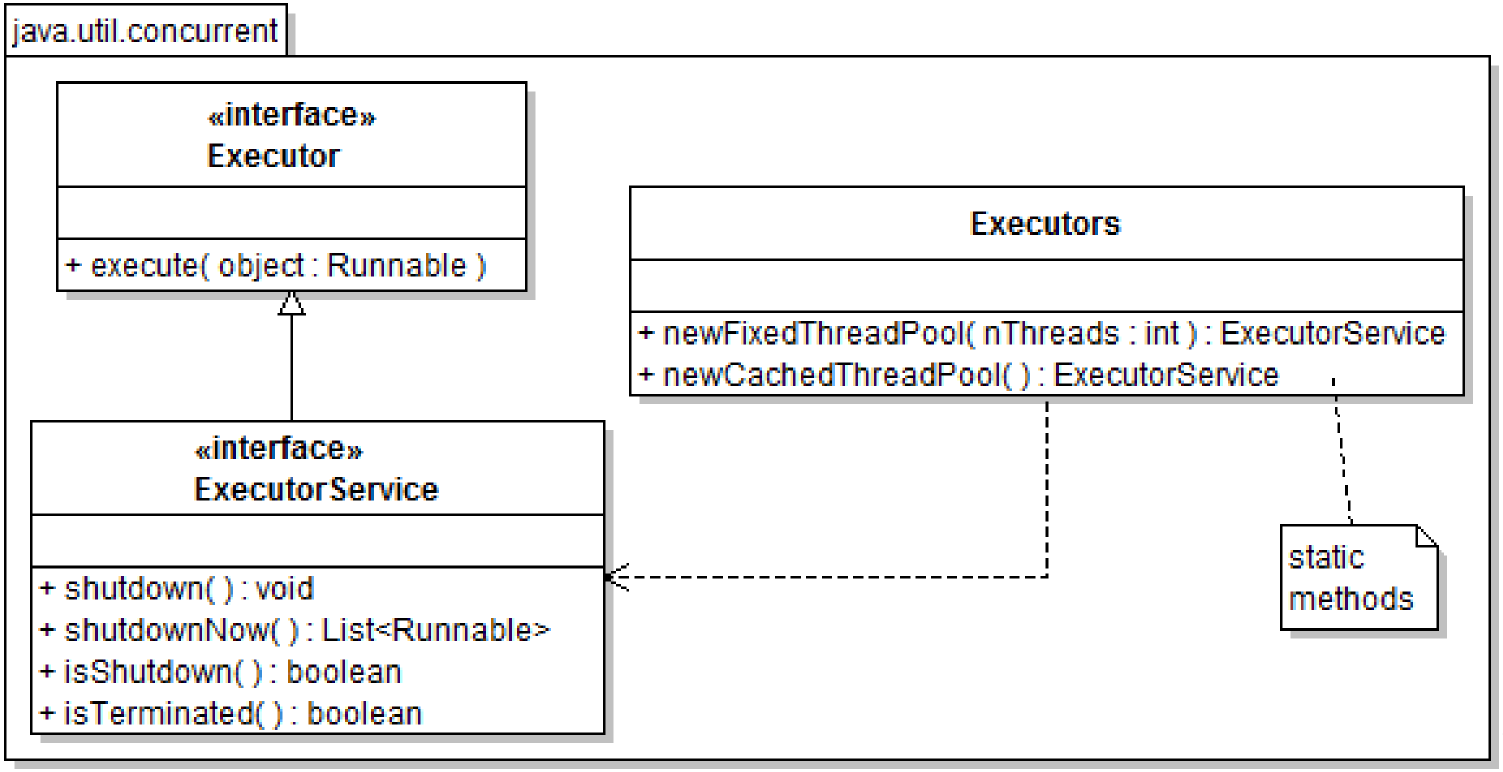
```
package oop06a;

public class Match {
    public static void main(String[] args) {
        Thread aThread;
        aThread = new Thread( new SoccerFan("G.S.", "Rulez!") );
        aThread.start( );
        aThread = new Thread( new SoccerFan("G.S.", "is the champ!") );
        aThread.start( );
        aThread = new Thread( new SoccerFan("F.B.", "is no.1!") );
        aThread.start( );
        aThread = new Thread( new SoccerFan("F.B.", "is the best!") );
        aThread.start( );
    }
}
```

2. Create an object of your class

4. Start the thread by using Thread.start

INTRODUCTION TO MULTITHREADING

- Thread pools:
  - Running a small number of tasks in separate threads is acceptable.
  - But do not forget that actual processing units in a typical CPU is rather low, and creating a thread has also a processing cost.
  - Therefore, if you are to execute a large number of tasks, you should use a thread pool instead.
  - Java provides the following interfaces and classes for this purpose:


java.util.concurrent
«interface»
Executor
+ execute(object : Runnable)
«interface»
ExecutorService
+ shutdown( ) : void
+ shutdownNow( ) : List<Runnable>
+ isShutdown( ) : boolean
+ isTerminated( ) : boolean
Executors
+ newFixedThreadPool(nThreads : int) : ExecutorService
+ newCachedThreadPool( ) : ExecutorService
static
methods

INTRODUCTION TO MULTITHREADING

- java.util.concurrent.ExecutorService:
  - public void shutdown( ) :
    - Shuts down the executor, but allows the tasks currently in the pool to be completed. New threads are not accepted to the pool.
    - We need to use this method for a safe ending.
  - public List<Runnable> shutdownNow( )
    - Shuts down immediately, stops the unfinished threads and returns them in a list.
  - public boolean isShutdown( ):
    - Returns true if the executor is shut down.
  - public boolean isTerminated( ):
    - Returns true if all the tasks in the pool are terminated.
    - Can be used in the main method for waiting the threads to be finished
- java.util.concurrent.Executor:
  - public void execute(Runnable object): Executes the given task
- java.util.concurrent.Executors:
  - public static ExecutorService newFixedThreadPool(nThreads : int)
    - Creates a thread pool that reuses a fixed number of threads
  - public static ExecutorService newCachedThreadPool( )
    - Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available

INTRODUCTION TO MULTITHREADING

- Let's modify our previous example to be run in a pool.
  - The SoccerFan class will not be changed.
  - Try using a fixed pool with different sizes!

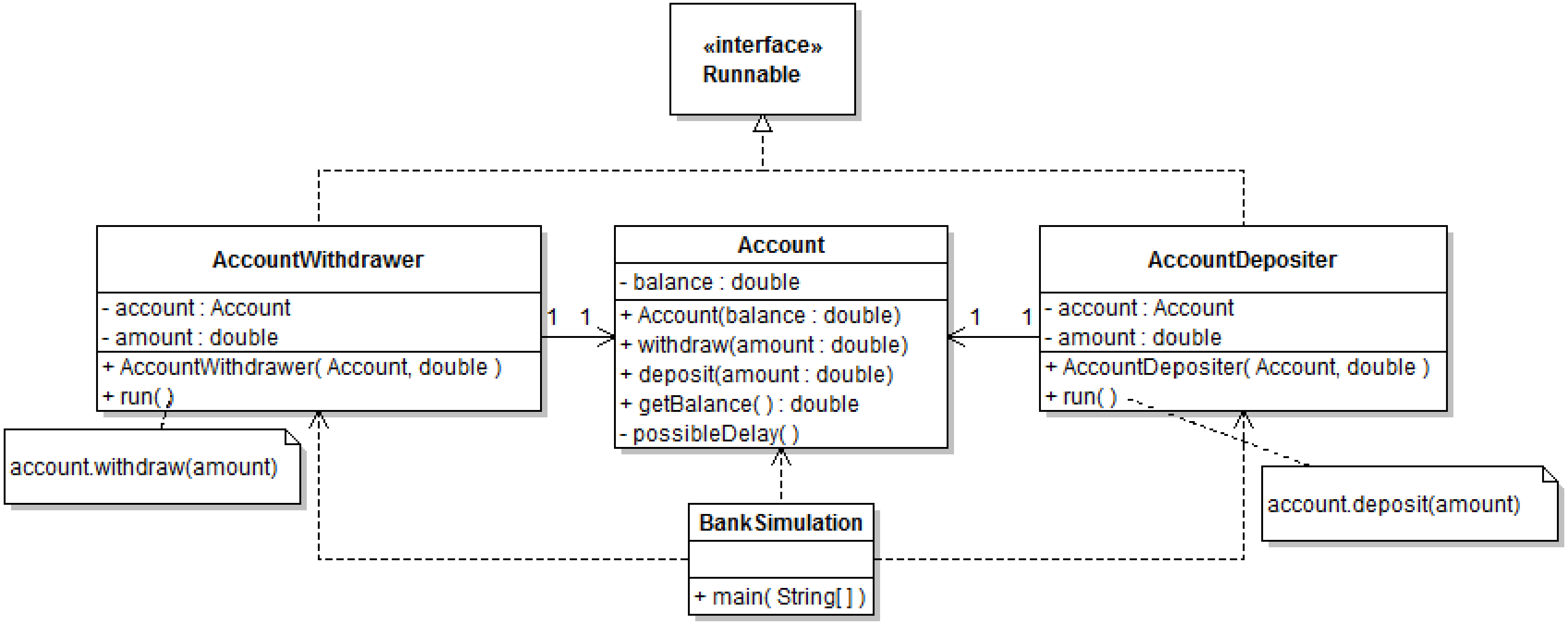
```
package oop06a;
import java.util.concurrent.*;
public class MatchWithPool {
    public static void main(String[] args) {
        ExecutorService pool = Executors.newCachedThreadPool( );
        pool.execute( new SoccerFan("G.S.", "Rulez!") );
        pool.execute( new SoccerFan("G.S.", "is the champ!") );
        pool.execute( new SoccerFan("F.B.", "is no.1!") );
        pool.execute( new SoccerFan("F.B.", "is the best!") );
        pool.shutdown( );
    }
}
```

INTRODUCTION TO MULTITHREADING

- Race condition:
  - In most practical multithreaded applications, two or more threads need to share access to the same data.
  - What happens if two threads have access to the same object and each calls a method that modifies the state of the object?
    - As you might imagine, the threads can step on each other’s toes!
    - Depending on the order in which the data were accessed, corrupted objects can result.
    - Such a situation is often called a race condition.

INTRODUCTION TO MULTITHREADING

- Thread synchronization is needed to avoid race conditions.
  - Consider the following example:



- This class diagram is descriptive enough, however, let’s write the code and execute it.

INTRODUCTION TO MULTITHREADING

```
package oop06b;
public class Account {
    private double balance;
    public Account(double balance) { this.balance = balance; }
    public double getBalance() { return balance; }
    public void withdraw( double amt ){
        double curBal = getBalance();
        possibleDelay( );
        balance = curBal - amt;
    }
    public void deposit( double amt ){
        double curBal = getBalance();
        possibleDelay( );
        balance = curBal + amt;
    }
    private void possibleDelay( ) {
        try { Thread.sleep(5); }
        catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

INTRODUCTION TO MULTITHREADING

```
package oop06b;
public class AccountDepositer implements Runnable {
    private Account account;
    private double amount;
    public AccountDepositer(Account account, double amount) {
        this.account = account; this.amount = amount;
    }
    public void run() {
        account.deposit(amount);
    }
}

package oop06b;
public class AccountWithdrawer implements Runnable {
    private Account account;
    private double amount;
    public AccountWithdrawer(Account account, double amount) {
        this.account = account; this.amount = amount;
    }
    public void run() {
        account.withdraw(amount);
    }
}
```

INTRODUCTION TO MULTITHREADING

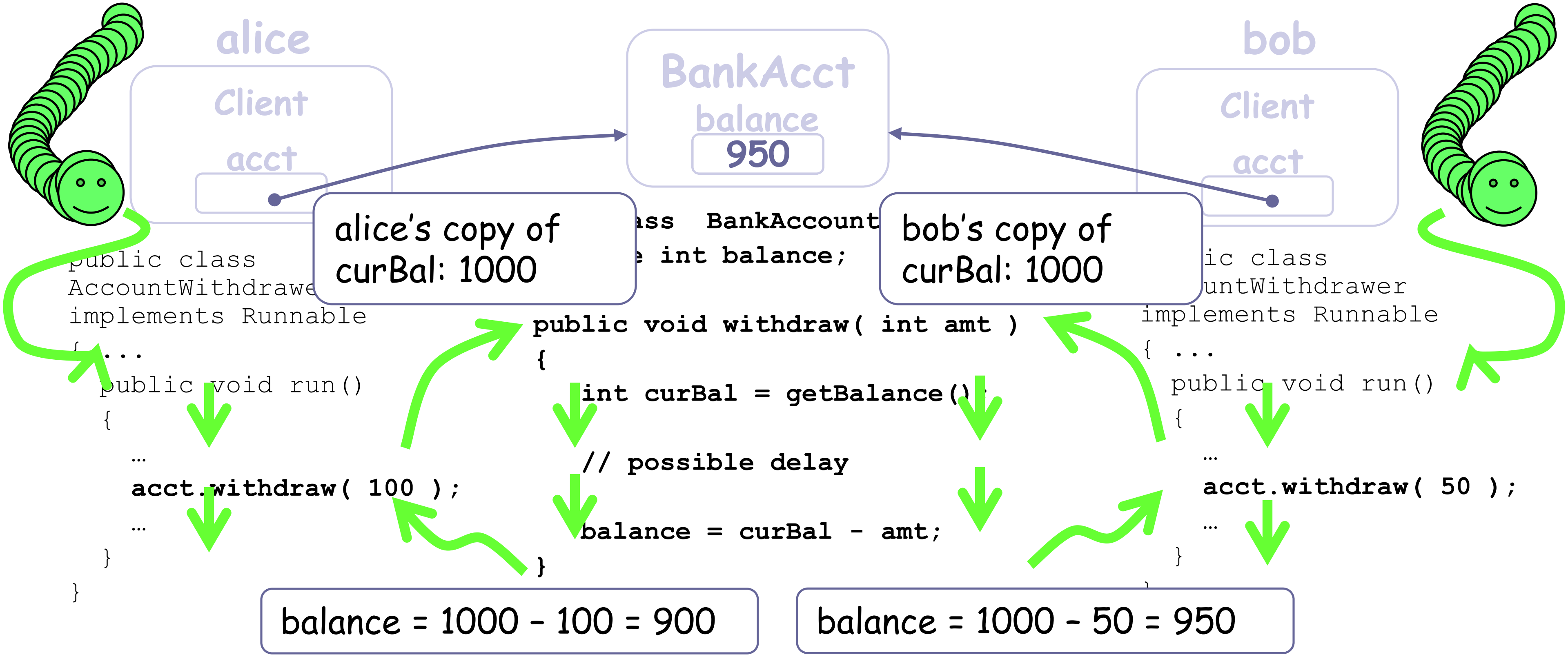
```java
package oop06b;
import java.util.concurrent.*;
public class BankSimulation {
    public static void main(String[] args) {
        Account anAccount = new Account(0);
        System.out.println("Before: "+anAccount.getBalance());
        ExecutorService executor = Executors.newCachedThreadPool( );
        for( int i = 0; i < 100; i++ ) {
            AccountDepositer task=new AccountDepositer(anAccount,1);
            executor.execute(task);
        }
        for( int i = 0; i < 50; i++ ) {
            AccountWithdrawer task=new AccountWithdrawer(anAccount,1);
            executor.execute(task);
        }
        executor.shutdown();
        while( !executor.isTerminated() );
                System.out.println("After: "+anAccount.getBalance());
    }
}
```

- What did you expect? What did you get?

INTRODUCTION TO MULTITHREADING

- What can happen if two threads tried to withdraw from a BankAccount at the same time?

*note: each thread has its own copy of local variables and parameters, but *fields* are shared between threads



```
public class
AccountWithdrawe
implements Runnable
{ ...
  public void run()
  {
    …
    acct.withdraw( 100 );
    …
  }
}
```

alice's copy of
curBal: 1000

```
ass  BankAccount
e int balance;

public void withdraw( int amt )
{
  int curBal = getBalance();

  // possible delay

  balance = curBal - amt;
}
```

bob's copy of
curBal: 1000

```
ic class
untWithdrawer
implements Runnable
{ ...
  public void run()
  {
    …
    acct.withdraw( 50 );
    …
  }
```

balance = 1000 - 100 = 900

balance = 1000 - 50 = 950

INTRODUCTION TO MULTITHREADING

- How can we prevent such a race?
  - We determine the methods which can lead to a race and label them with the keyword synchronized.
  - Only one thread can execute a synchronized mehod, others wait.

```
package oop06c;
public class Account {
    private double balance;
    public Account(double balance) { this.balance = balance; }
    public synchronized void withdraw( double amt )  {
        double curBal = getBalance();
        possibleDelay( );
        balance = curBal - amt;
    }
    public synchronized void deposit( double amt )   {
        double curBal = getBalance();
        possibleDelay( );
        balance = curBal + amt;
    }
    public double getBalance() { return balance; }
    public void possibleDelay( ) { /*same as the previous one */ }
}
```

INTRODUCTION TO MULTITHREADING

- Other classes stay the same.
- Output:

```
Before: 0.0
After: 50.0
```

- About the data structures and multithreading:
  - Remember the data structures section: Some data structures are thread-safe, i.e. synchronized
  - Vector<E> and Hashtable<K,V>
  - Use those data structures when multithreading is to be used.