BLM 2012 - OBJECT ORIENTED PROGRAMMING
February 2019
Associate Prof. Dr. Mehmet Sıddık AKTAŞ

GENERAL INFORMATION

SCORING

1st midterm: %20	Week 9: April 11st, 2019
2nd midterm: %20	Week 13: May 9th, 2019
Midterm makeup	Week 14: May 16th, 2019
Final exam: %40	Finals week **(TBA)**
Lab: %10	Begins in week 3: Feb. 28h, 2019
Project: %10	Due date and details **(TBA)**

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.

GENERAL INFORMATION

GROUPS

- Gr.1 Dr. Öğr. Üyesi Yunus Emre Selçuk
- Gr.2 Doç. Dr. Mehmet Sıddık Aktaş
- Pay attention to enter lectures and exams in your registered group

HIGHLIGHTS

- Labs:
  - Lectures will be given by instructors in lab hours by instructors for 2 weeks
  - When the lab schedule starts, lab activities and classroom example activities will alternate
  - Check lab assistant’s pages for updated information
    - (İbrahim Onur Sığırcı for 2018-2)
- Regulation:
  - A student with success note lower than 40 will fail a course with FF, whether s/he has taken that course before or not.
    - Interpretion: 40 cannot correspond to CC

GENERAL INFORMATION

COURSE OUTLINE

- General Outline of the Java Programming Language
- Objects and Classes
- UML Class Schemas
- Object State, Behaviour and Methods
- Object and Class Collaborations and Relations
- UML Interaction (Sequence) Diagrams
- Inheritance and Abstract Classes
- Interfaces and Multiple Inheritance
- Polymorphism, Method Overriding and Overloading

- 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

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

JAVA EXECUTION ENVIRONMENT

- Standard Edition (JSE):
  - Suitable for developing any kind of application except applications for mobile devices
- Micro Edition (JME):
  - Suitable for developing applications for mobile devices, smartphones, etc.
  - Contains a subset of libraries of JSE.
- Enterprise Edition (JEE):
  - Contains JSE and an application server software (App. server)
    - App. server gives several services to applications coded by JSE.
    - More complex applications such as multi-tiered applications, web services, etc. need these services.
    - Transaction support is one of these services.
  - The basic App. Server is named "Sun Java System Application Server".
  - But there are other compatible services as well:
    - IBM Websphere
    - BEA WebLogic
    - Apache Tomcat
    - …

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

JAVA EDITIONS

- The old and the new way of naming Java:

Developer Version (Old way)	Product Version (New way)
Java 1.0, 1.1	
Java 1.2	Java 2 Platform
Java 1.3	Java 2 SE 3 (J2SE3)
Java 1.4	J2SE4
Java 1.5	J2SE5
Java 1.6 {Sun}	Java Platform Standard Edition, version 6 (Java SE6 / JSE6)
Java 1.7 {Oracle}	Java Platform Standard Edition, version 7 (Java SE7 / JSE7)
Java 1.8 (preferred)	Java Platform Standard Edition, version 8 (Java SE8 / JSE8) Has LTS (Long Term Support) and language features not covered in this lecture
Java 1.9, 1.10	Short-term releases (~6 months each)
Java 1.11	Java SE11. Has LTS, language features and tools not covered in this lecture

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

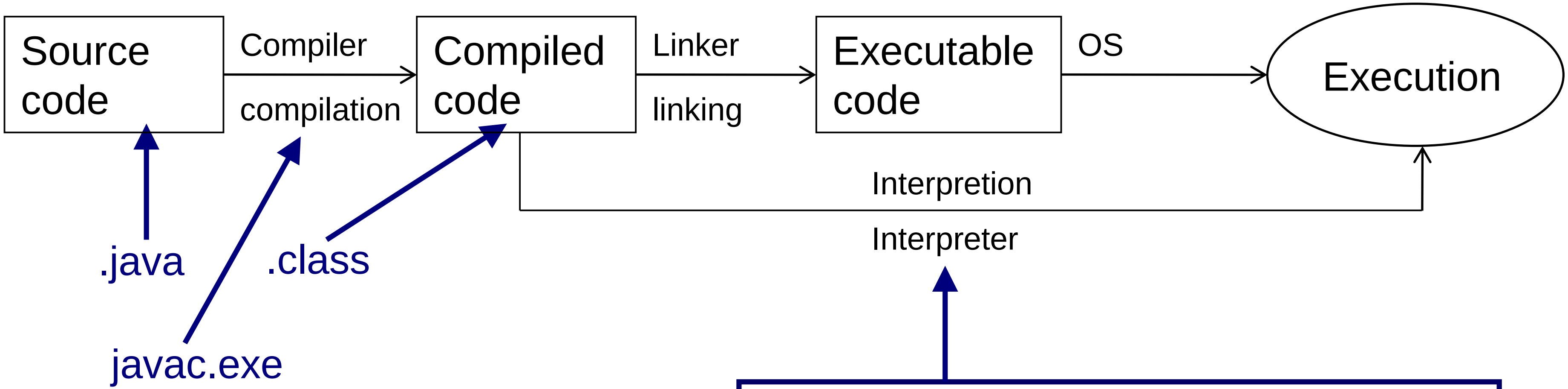
JAVA EDITIONS

- Versioning details:
  - JDK 1.8.0_202:
    - Java Version 8.0, update 202.
    - Update:
      - Small bug fixes and security improvements.
      - Updated multiple times a year.
- Where to obtain:
  - oracle.com/java (web site changes frequently!)
    - http://www.oracle.com/technetwork/java/javase/downloads/index.html
  - Select the correct version: x86(i586) vs. x64
  - Download and unzip the documentation as well.

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

JAVA EXECUTION ENVIRONMENT

- Java is an interpreted language



Source code

Compiler

compilation

Compiled code

Linker

linking

Executable code

OS

Execution

Interpretion

Interpreter

.java

.class

javac.exe

- JDK: Java Development Kit
  - Includes javac.exe
  - Optionally excludes IDE

- Java Virtual Machine (JVM)
  - java.exe/javaw.exe: for command line/GUI apps
  - is included in JRE (Java Runtime Environment)

- JDK is for developers and JRE is for end-users
  - JDK includes JRE

GENERAL OUTLINE OF THE JAVA PROGRAMMING LANGUAGE

FREE JAVA DEVELOPMENT TOOLS

- IDE: Integrated Development Environment
- Eclipse: http://www.eclipse.org
  - Downloaded separately
  - You need to install a plug-in (such as eUML2) for drawing UML schemas.
  - You need to install a plug-in for writing GUI applications.
  - No need to have administrator rights on the computer, just unzip it.
- NetBeans:
  - Download separately or optionally with JSE.
  - You need to install a plug-in (suggestions?) for drawing UML schemas.
  - Has built-in GUI editor.
  - Needs administrator rights for installation.

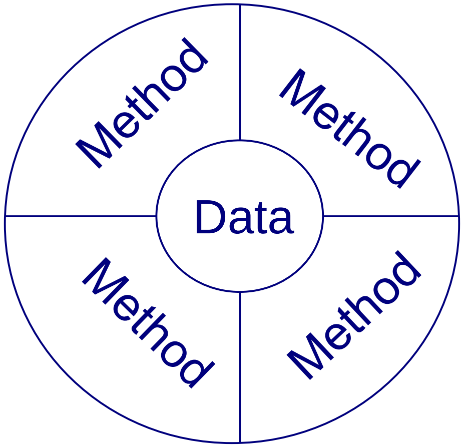
FREE UML MODELING TOOLS

- Violet UML: Lightweight, enough for this course.
- Argo UML

CLASSES, OBJECTS AND MEMBERS

OBJECT

- Object: The main programming element.
  - Contains attributes and tasks.
  - Object ≈ a real-world entity.
    - Similar to variables but Superman is similar to mere mortals, too!
  - Attributes of an object ≈ Data about this entity.
  - Tasks ≈ actions ≈ methods
    - Similar to functions but …
    - Each function can access:
      - Any attribute of an object and,
      - Any given parameter but …
      - … there are many rules!
      - Our purpose is to master these rules.



- Encapsulation: The data and the methods of an object cannot be separated.
  - Data is accessed through methods.

CLASSES, OBJECTS AND MEMBERS

CLASS

- A class is just a template which defines objects.
  - The program is coded as classes, but the real work is done by the objects.
  - You may think a class as a cookie cutter and think objects as cookies!

```
class myClass {
   //program code
}
```

CLASSES, OBJECTS AND MEMBERS

OBJECTS AND CLASSES

- An example object: A particular car.
  - Attributes: Model, license plate number, color, etc.
    - Usually, one of the attributes of an object is determined as its logical unique identifier (UID).
    - Such as the plate number of a car.
  - Actions: Query a car about its license plate number, to sell this car, etc.
- An example class: Automobile.
  - A program code which defines the attributes and methods of cars.
- You can create any number of classes from any different classes within an object oriented program.

CLASSES, OBJECTS AND MEMBERS

OBJECTS AND CLASSES

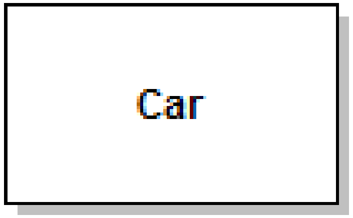
- The attributes of an object can be conceptually divided into two groups:
  - Primitives: One unit of information such as integer numbers, real numbers and boolean values.
  - Non-primitives: Any number of objects from any number of classes.

CLASSES, OBJECTS AND MEMBERS

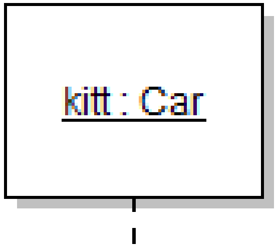
TERMINOLOGY AND REPRESENTATION

- OOP Terminology:
  - Data = **Member field** = field = attribute
  - State: Set of values of all attributes of an object
  - Task = Action = function = Method = **Member method**
  - **Members** of a class/object = Methods + fields
  - **Class** = type.
  - If o is an object of class C, we can also say that o is an **instance of** S.

- UML Representation:



A class shown in a class diagram



An object shown in a sequence diagram

CLASSES, OBJECTS AND MEMBERS

TERMINOLOGY AND REPRESENTATION

- There are two kinds of UML interaction diagrams:
  1. Sequence diagrams
  2. Collaboration diagrams

CLASSES, OBJECTS AND MEMBERS

EACH OBJECT IS A DIFFERENT INDIVIDUAL!

- Consider two objects of the same type:
  - Although both have the same type of attributes, the values of these attributes will be different = The **state**s of these objects will be different.
  - Even if you create two objects having the same state, these two objects will be represented in different areas of the memory.
    - In Java, the JVM creates a unique identifer for this purpose. This process, as well as the other memory management processes, are transparent to the programmer.
      - so transparent that you cannot interfere with
- Example: Any two cars cruising in the street.
  - Some attributes: Model, color, license plate.
  - The models and the colors of these cars will be different.
  - Even if you see the same yellow Anadol STC's, their license plates will be different.
    - Even if there is a conteirfeiting in effect so that their license plates are the same, their drivers will be different!

CLASSES, OBJECTS AND MEMBERS

EACH OBJECT IS A DIFFERENT INDIVIDUAL!

- Two different objects will give different answers to the same message, even if they are of the same type.
    - Why? Because their states will be different.
    - Moreover, you can give different parameters to the same method.

CLASSES, OBJECTS AND MEMBERS

SENDING MESSAGES TO OBJECTS

- Why do we send a message to an object?
  - In order to have this object to do something
  - To access a member of this object

MEMBER ACCESS

- We access a member field of an object in order to:
  - Change its value (setting)
  - Read its value (getting)

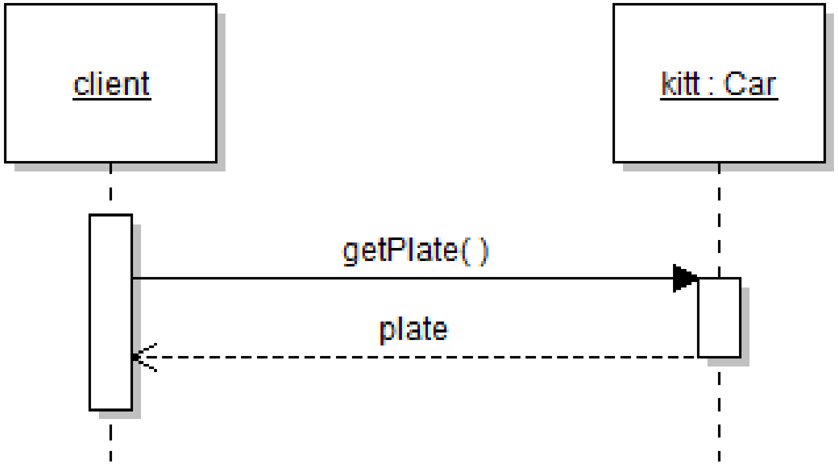
- We access a member method of an object in order to :
  - Run a method, optionally with some parameters
  - Calling a method is similar to calling a function in C.
    - But remember: Unless otherwise, a method of an object works with the members of this object.
      - How come otherwise?
        - Wait until you learn the different kinds of relationships between objects.

CLASSES, OBJECTS AND MEMBERS

TERMINOLOGY AND REPRESENTATION

- OOP Terminology:
  - An object accesses a member of another object by sending a message to that other object
  - An object oriented program executes as passed messages between objects.

- UML Representation:



- Code representation:

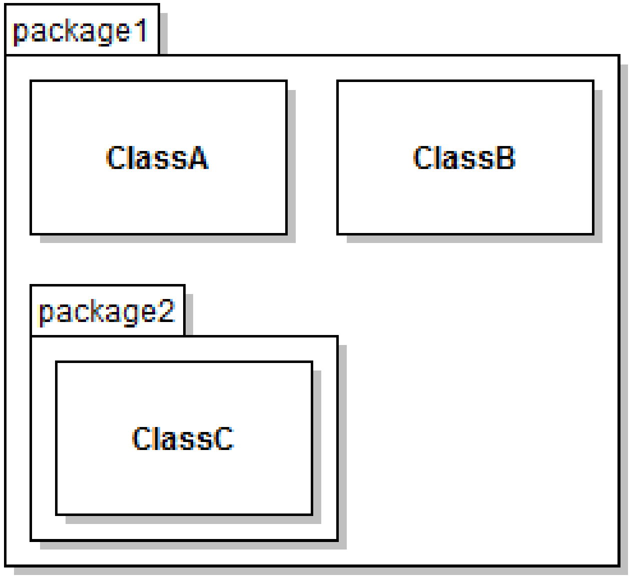
```
kitt.getPlate();
```

- Meaning of this figure:
  - There is an object named client
  - The class of the client is irrevelant
  - There is an object named kitt
  - Car is the class of the object kitt
  - The class Car has a method named getPlate
  - The client sends the message getPlate to the object kitt
  - the object kitt returns its license plate as the answer to the message

CLASSES, OBJECTS AND MEMBERS

PACKAGES

- Classes can be grouped together in abstract containers named packages
- The aim is to group such classes that can be used for a particular common purpose.



- Adding classes in a package to our code:

```
import package1.ClassA;
import package1.*;
import package1.package2.*;
```

  - Classes of package2 are not included when package1 is imported.

- Different classes in different packages may have the same name,
  - yet no conflicts arise.

```
java.io.File
com.fileWizard.File
```

- File path hierarchy must reflect the package hierarchy:

```
com.fileWizard.File -> com\fileWizard\File.java
```

CLASSES, OBJECTS AND MEMBERS

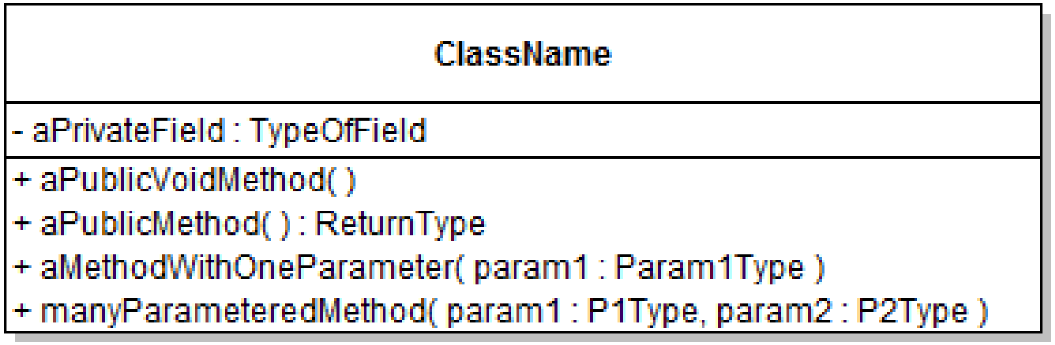
VISIBILITY RULES AND INFORMATION HIDING

- An object can access all of its members and all members of other objects belonging to the same class (type).
- However, we can hide some members of an object so that they cannot be accessed from objects of different types.
- The information hiding principle:
  - We hide the members that are related with the inner workings of an object from objects of different types.
  - So that an object does not need to know the internal details of another object in order to use that other object.
- Example: It is sufficient to know the universal signs of power, volume and channel switching keys on a remote in order to watch TV.
  - You don't need to know that this TV has a device named cathod tube in it.
  - Moreover, the users need not to be re-educated for using TVs built with new technologies such as LCD, plasma, etc.
- Example: Your friend wants to lend some money from you.
  - You either open your purse and give him/her that money or not.
  - You don't have to tell anything about your salary or your PIN number to your friend!

CLASSES, OBJECTS AND MEMBERS

VISIBILITY RULES AND INFORMATION HIDING

- Access modifiers (Visibility rules):
  - public: There are no access restrictions to public members
  - private: Objects of different types cannot access each other's private members
- UML representation:



ClassName

- aPrivateField : TypeOfField

+ aPublicVoidMethod( )
+ aPublicMethod( ) : ReturnType
+ aMethodWithOneParameter(param1 : Param1Type)
+ manyParameteredMethod(param1 : P1Type, param2 : P2Type)

- Moreover (you are not responsible from those in this class):
  - protected: #
    - Related with inheritance (visible to package and subclasses)
  - package: ~
    - visible to package
    - Default rule in Java

CLASSES, OBJECTS AND MEMBERS

VISIBILITY RULES AND INFORMATION HIDING

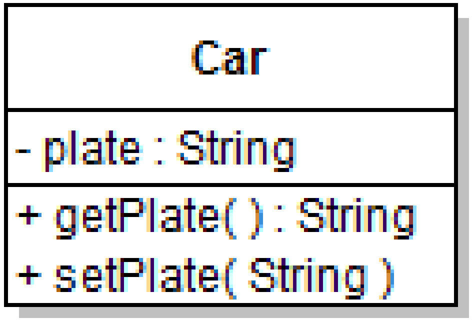
- In practice, the information hiding principle cannot be applied in a perfect way.
  - A change in code of a class not only affects that class but other classes that are related with that class as well.
  - The further you comply with this principle, the easier your coding overhead becomes for completing this change as the number of the affected classes will reduce.
- In order to comply with the information hiding principle:
  - Member fields are defined as private, and…
  - …the necessary access methods are defined as public.
  - At least 5 points for each question will be deduced if you don't comply!

- Access methods (accessors):
  - Setter method: Used for changing the value of a member field of an object.
  - Getter method: Used for reading the value of a member field of an object.
  - Naming convention: getMember, setMember

CLASSES, OBJECTS AND MEMBERS

VISIBILITY RULES AND INFORMATION HIDING

- Example:



Car

- plate : String

+ getPlate( ) : String

+ setPlate(String)

- You can easily change the permissions to member fields. For example:
    - If you need to restrict the modification of the license plate, remove the setPlate method from code.
    - If you need to permit only the classes in the same package to make this modification, change the visibility of setPlate to package.

CLASSES, OBJECTS AND MEMBERS

SPECIAL CASES OF MEMBERS

- Static member fields:
  - The state of each object, even though they are of the same class, is different.
  - However, in some cases, you may need to have **all** objects of a particular type to **share** a common member field.
  - In this case, you define this member field with the **`static`** keyword.
  - Static members are accessed via the class name such as **`ClassName.memberName`**, not via the objects.
  - Example: Each automobile has 4 tires.
- Static member methods:
  - Two different objects of the same type answer the same message differently.
  - However, in some cases, you may need to have **all** objects of a particular type to **share** a common behavior.
  - In this case, you define this member method with the **`static`** keyword.
  - You may only use static members of an object within a static method of this object.
  - They are accessed via the class name, i.e. **`ClassName.aMethod()`**

CLASSES, OBJECTS AND MEMBERS

SPECIAL CASES OF MEMBERS

- Final member fields:
    - You may need the value of a member field to stay constant.
    - In this case, you define this member field with the **`final`** keyword
    - You may assign a value to a final member of an object only once
        - This assignment is usually done when that object is created.
    - For example, the chassis number of a car is etched onto it when it is produced in the factory and it cannot be changed afterwards.
- Final member methods:
    - These cannot be overridden (inheritance will be taught later).

POINTS TO CONSIDER

- A member can be both final and static at the same time.
- Do not confuse final and static with each other:
    - Final: Only once
    - Static: Shared usage
- Shown in UML class schemas as: aMember : Type {final,static}

CLASSES, OBJECTS AND MEMBERS

CONSTRUCTORS AND FINALIZERS

- Constructor Method:
  - This method is executed explicitly by the programmer when an object is to be created.
  - Constructors are used for assigning the initial values of the member fields of an object.
  - We will pay a significant attention to constructors in this class.
- Finalizing method:
  - This method is executed implicitly by JVM when an object is to be destroyed.
  - The method name is finalize
    - It takes no parameters and it does not return anything.
  - Unlike C/C++, Java programmers mostly need not to handle memory management.
  - As a result, we will not study finalizer methods any more in this class.

CLASSES, OBJECTS AND MEMBERS

CONSTRUCTOR METHODS

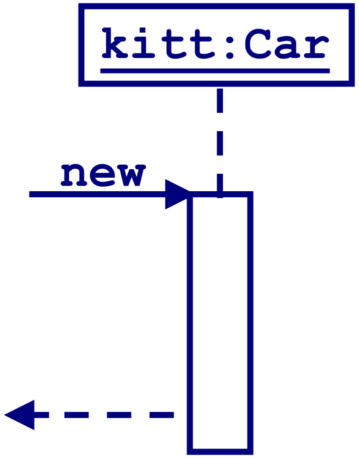
- Rules for constructors:
  - They are public.
  - Their name is the same with the class
  - Although they are used to create an object,
    - You do not issue a return command within constructor body and
    - you do not give a return type to the constructor method.
  - It's the best place to assign values to final member fields.
  - They are used with the **`new`** keyword.

```
kitt = new Car();
```

CLASSES, OBJECTS AND MEMBERS

CONSTRUCTOR METHODS

- Defining an object is not enough to use it, you need to ***instantiate*** it by executing its constructor.
- UML Representation:



- Code representation1: As a member field

```
public class AClass {
   private Car kitt;

   someMethod( ) {
       kitt = new Car();
   }
}
```

- Code representation2: As a temporary object/variable

```
public class AnotherClass {
   someMethod( ) {
       Car kitt = new Car();
   }
}
```

CLASSES, OBJECTS AND MEMBERS

CONSTRUCTOR METHODS

- Default constructor:
  - The constructor without parameters.
  - JVM implicitly and automatically defines a default constructor if the programmer does not code any constructor.
- Constructors with parameters:
  - The parameters are used for assigning initial values.
  - If the programmer explicitly code a constructor with parameters, the default constructor is not automatically created by JVM.
    - In this case, it's up to the programmer to code a default constructor, if one required.
    - However, business logic often requires parametered constructors and forbids default constructors.
- A class can have more than one constructors having different types and numbers of parameters.
  - This is called constructor **overloading**.
  - Regular methods also can be overloaded.

CODING AN OBJECT ORIENTED PROGRAM

CONTROL FLOW

- Control flow is the order of execution of program codes.
  - In the lowest level, a computer program consists of various commands that are executed in a particular order.
  - The order that these commands are written and the order that they are executed are not necessarily the same.
    - In fact, especially in OOP, these two orderings are almost always quite different than each other.
  - Luckily, the starting point of this control flow is easier to determine.

CODING AN OBJECT ORIENTED PROGRAM

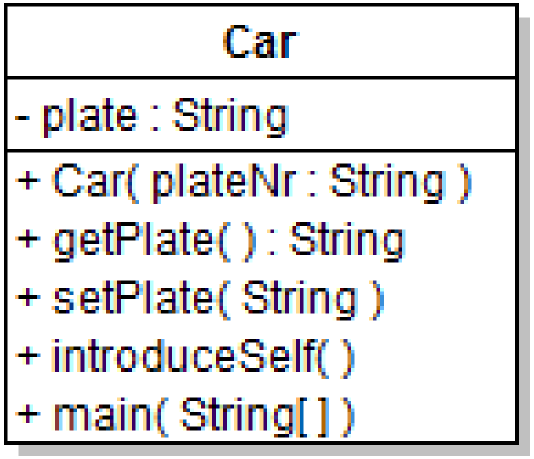
BEGINNING OF THE CONTROL FLOW

- The control flow of a program should have a starting point.
    - This point is a static method, named main, within a particular class that is determined by the programmer.
        - public static void main(String[ ] args)
            - The array args is used for passing initial parameters to the program from the command line.
            - static: It cannot be otherwise, because:
            - No object is created at the beginning of the control flow.
    - The task of the main method is to create the initial object(s) and to begin the execution of the program.
        - Remember, an OO program consists of messages sent between objects.
    - The existance of a main method in a class does not imply that this method will always be used.
- Terminology: Block/body: A piece of code having multiple instructions.
    - Shown between curly braces: { }

CODING AN OBJECT ORIENTED PROGRAM

CREATING YOUR OWN CLASS AND OBJECTS

- UML representation (class diagram)



Car

- plate : String

+ Car(plateNr : String)
+ getPlate( ) : String
+ setPlate(String)
+ introduceSelf( )
+ main(String[ ])

- Draw the class schema first.
- Then map the schema and the code
- Pretty printing, camel casing …

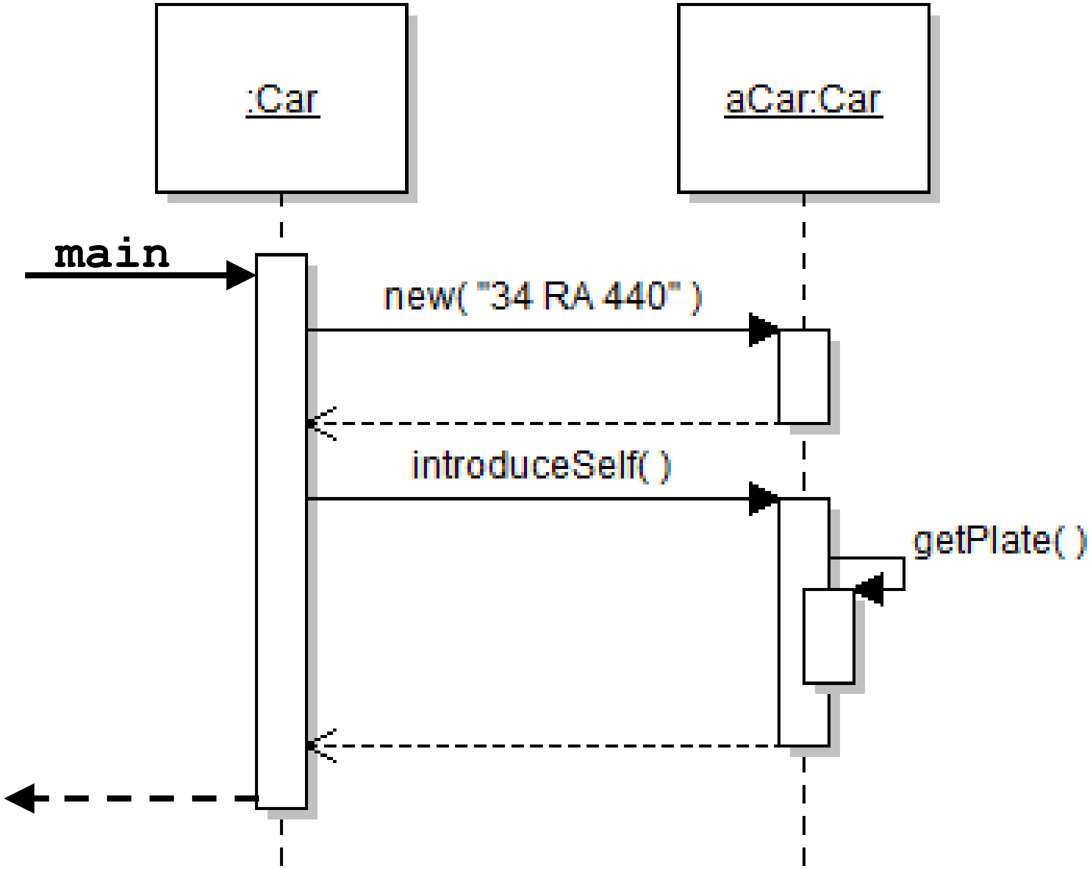
- Source code (implementation)

```
package nyp01a;
public class Car {
   private String plate;
   public Car( String plateNr ) {
      plate = plateNr;
   }
   public String getPlate() {
      return plate;
   }
   public void setPlate(String plate) {
      this.plate = plate;
   }
   public void introduceSelf( ) {
      System.out.println( "My plate: " + getPlate() );
   }
   public static void main( String[] args ) {
      Car aCar;
      aCar = new Car( "34 RA 440" );
      aCar.introduceSelf( );
   }
}
```

CODING AN OBJECT ORIENTED PROGRAM

UML REPRESENTATION

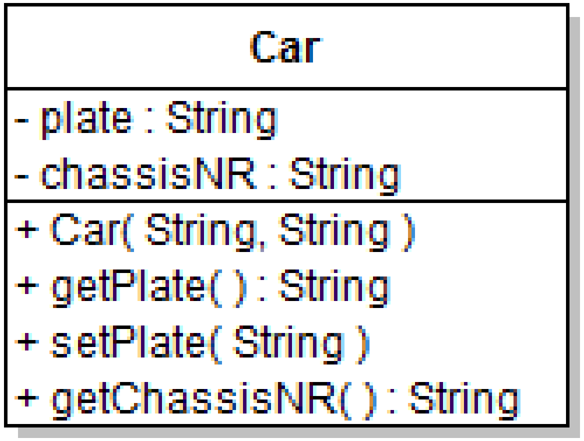
- Representation of the main method in the sample code by a sequence diagram, which is a kind of interaction diagram.
  - Pay utmost attention to the ordering and alignment of the arrows!



CODING AN OBJECT ORIENTED PROGRAM

CREATING YOUR OWN CLASS AND OBJECTS

- Another version of the class Car :



Car

- plate : String
- chassisNR : String

+ Car(String, String)
+ getPlate( ) : String
+ setPlate(String)
+ getChassisNR( ) : String

```
package nyp01b;
public class Car {
   private String plate;
   private String chassisNR;
   public Car( String plateNr, String chassisNR ) {
      plate = plateNr;
      this.chassisNR = chassisNR;
   }
   public String getPlate() {
      return plate;
   }
   public void setPlate(String plate) {
      this.plate = plate;
   }
   public String getChassisNR( ) {
      return chassisNR;
   }
}
```

- This second version of the class Car does not have a main method.
  - Therefore, it cannot be run and tested alone.
  - We need to code another class with a main method with these purposes (will be shown later).

CODING AN OBJECT ORIENTED PROGRAM

CREATING YOUR OWN CLASS AND OBJECTS

- Pay attention to the constructor:
  - In real world, every car must have a license plate AND a chassis number.
  - Therefore, both fields must be initialized in the same constructor having two parameters.
  - The code at the left is right, the code at the right is wrong.

```
public class Car {
   private String plate;
   private String chassisNR;
   public Car( String plateNr,
             String chassisNR ) {
      plate = plateNr;
      this.chassisNR = chassisNR;
   }
   /* Rest of the code */
}
```

```
public class Car {
   private String plate;
   private String chassisNR;
   public Car( String plateNr ) {
      plate = plateNr;
   }
   public Car(String chassisNR ) {
      this.chassisNR = chassisNR;
   }
   /* Rest of the code */
}
```

- Compile error vs. bug:
  - The code at the right does not compile. If it had, it's logic would be wrong (buggy).
- In real world, the chassis number of a car never changes. Therefore, we didn't code the getter method of that field. If we had, we would introduce another bug! (final fields can be mentioned shortly)

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

PRIMITIVES AND WRAPPERS

- Primitive type: One unit of information (non-class).
- Wrapper: A class having one primitive member field 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

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

PRIMITIVES AND WRAPPERS

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

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

- Other primitives:

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

- We will also give more details on the non-primitive type String soon.

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

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

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

PRIMITIVES AND WRAPPERS

- You can think of a wrapper as a class with only one member field of a primitive type that it wraps/boxes.
- 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)

```
Integer sarma1 = 1, sarma2 = 7;
//could use sarma1 = new Integer( 1 ); but that constructor is deprecated,
//i.e it will be removed from the language in a future version
System.out.println("Sonuç1:"+sarma1.compareTo(sarma2)); //-1
```

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

STRING CLASS

- Some methods of the class String
  - int length()
  - int compareTo(String anotherString)
  - int compareToIgnoreCase(String str)
- System.out.println(String)
  - print / println
- Example:

```
package nyp01c;
public class StringOps01 {
   public static void main( String args[] ) {
       String strA, strB;
       strA = "A string!";
       strB = "This is another one.";
       System.out.println(strA.compareTo(strB));
   }
}
```

- Output of the example: −

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

STRING CLASS (Continued)

- Some methods of the class String (continued):
    - boolean contains(String anotherString)
    - String toUpperCase( )
    - String toLowerCase( )
    - Note: toUpper/LowerCase methods do not change the state of the object.
    - Considering that note, what will the output of the code given in the next slide will be?

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

STRING CLASS (Continued)

```java
package nyp01c;
public class StringOps02 {
   public static void main( String args[] ) {
       String strA = "İstanbul", strB = "Yıldız";
       System.out.println(strA.contains(strB));
       strB = "tan";
       System.out.println(strA.contains(strB));
       strB.toUpperCase();
       System.out.println(strB);
       System.out.println(strA.contains(strB));
       strB = strB.toUpperCase();
       System.out.println(strB);
       System.out.println(strA.contains(strB));
   }
}
```

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MATH CLASS

- This class has static methods for common mathematical functions.
  - public static double Math.random( )
    - Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0
    - Example code:

```
package nyp01c;
public class MathOps01 {
    public static void main(String[] args) {
        double value = Math.random();
        System.out.println("The generated random value is: " + value);
    }
}
```

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MATH CLASS (Continued)

- This class has static methods for common mathematical functions (cont'd:)
  - variations of public static <primitive> Math.abs(<primitive> a)
    - returns the absolute value of parameter a where <primitive> is any primitive type, i.e.
    - public static double Math.abs(double a)
  - variations of public static <primitive> Math.max(<primitive> a, b)
    - returns the value of the greater of the two parameters where <primitive> is any primitive type, i.e.
    - public static double Math.max(double a, double b)
  - variations of public static <primitive> Math.min(<primitive> a, b)
    - returns the value of the smaller of the two parameters where <primitive> is any primitive type, i.e.
    - public static double Math.min(double a, double b)

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MATH CLASS (Continued)

- This class has static methods for common mathematical functions (cont'd:)
  - public static double Math.ceil(double a)
    - returns the upwards-rounded value of a (i.e. 3.1 → 4.0)
  - public static double Math.floor(double a)
    - returns the downwards-rounded value of a (i.e. 3.9 → 3.0)
  - public static double Math.round(double a)
    - returns the correctly-rounded value of a (i.e. 3.5 → 4.0, 3.1 → 3.0)
  - public static double Math.sqrt(double a)
    - returns the correctly rounded positive square root of a

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MORE ON RANDOM VALUES

- All computer systems rely on pseudo-random number generators.
  - It is modeled by java.util.Random class
  - If Math.random( ) is used directly, JRE automatically generates a Random object and uses it in the entire lifetime of the JVM
  - Random class has some useful non-static methods to obtain random values of desired primitives:
    - public boolean nextBoolean( )   returns [false, true]
    - public double nextDouble( )   returns [0.0, 1.0)
    - public float nextFloat ( )   returns [0.0, 1.0)
    - public int nextInt( )   returns $(-2^{32}, 2^{32})$
    - public int nextInt(int bound)   returns [0, bound)
    - public long nextLong( )   returns $[0, 2^{48})$

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

MORE ON RANDOM VALUES

- Example code:

```
package nyp01c;
import java.util.*;
public class RandomOps {
   public static void main(String[] args) {
      Random generator = new Random();
      int intVal = generator.nextInt();
      System.out.println("I have got " + Math.abs(intVal) + " pebbles.");
      int bounded = generator.nextInt(11);
      System.out.println("I have painted my " + bounded + " fingers.");
   }
}
```

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

COMMAND LINE I/O

- Output with System.out object:
  - The out member of System is a public and static member
    - The object out can therefore be used directly.
  - Methods for command line output:
    - printLn, print: We have learned those
    - printf: Used just as the C programmers know

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

COMMAND LINE I/O

- Input operations with the java.util.Scanner class: with JDK 5.0 and later!
  - Initialization: Scanner in = new Scanner(System.in);
  - System.in : A public static member of type java.io.InputStream.
  - Methods for obtaining input (one element at a time):
    - String nextLine()
    - int nextInt()
    - float nextFloat()
    - …

```
package nyp01c;
import java.util.Scanner;
public class ConsoleIOv1 {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      System.out.print("What is your name? ");
      String name = in.nextLine();
      System.out.print("How old are you? ");
      int age = in.nextInt();
      System.out.println("Hello, " + name +
      ". Next year, you'll be " + (age + 1) + ".");
      in.close();
   }
}
```

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

COMMAND LINE I/O

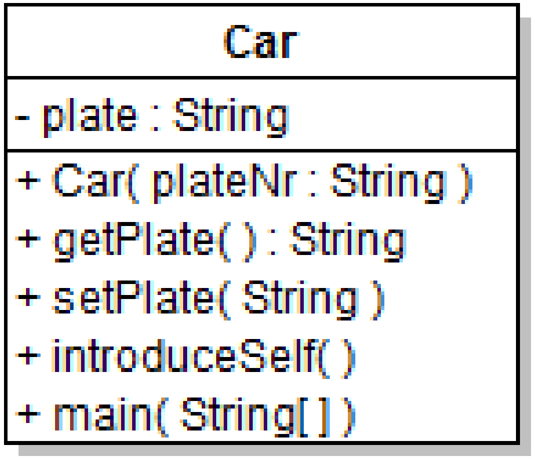
- A bug in the Scanner class:
  - If you get input for a string after getting input for a primitive by using nextInt, nextFloat, etc., that string goes to void!
  - As a workaround, issue an empty nextLine command in such cases.

```
package nyp01c;
import java.util.Scanner;
public class ConsoleIOv2 {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      System.out.print("How old are you? ");
      int age = in.nextInt();
      in.nextLine(); //workaround for the bug
      System.out.print("What is your name? ");
      String name = in.nextLine();
      System.out.println("Hello, " + name +
      ". Next year, you'll be " + (age + 1) + ".");
      in.close();
   }
}
```

FUNDAMENTAL DATA REPRESENTATION AND OPERATIONS

COMMAND LINE I/O

- Let's change the main method so that the license plate of the car is obtained from the user:



Car

- plate : String

+ Car(plateNr : String)
+ getPlate( ) : String
+ setPlate(String)
+ introduceSelf( )
+ main(String[ ])

```
package nyp01d;
import java.util.*;
public class Car {
   private String plate;
   public String getPlate() {
      return plate;
   }
   public void setPlate(String plate) {
      this.plate = plate;
   }
   public Car( String plateNr ) {
      plate = plateNr;
   }
   public void introduceSelf( ) {
      System.out.println( "My plate: " + getPlate() );
   }
   public static void main( String[] args ) {
      Car aCar;
      Scanner input = new Scanner( System.in );
      System.out.print("Enter a license plate: ");
      aCar = new Car( input.nextLine() );
      aCar.introduceSelf( );
      input.close();
   }
}
```

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

```java
package nyp01e;
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 = 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 = 1;
          Integer wrap2 = 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 nyp01e;
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 = 3;
        Integer sarma2 = 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:

```
package nyp01e;
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
```

ALTERING THE CONTROL FLOW

- The structures you are familiar with since BBG2 also exists in Java with similar syntax.
- A short summary is given below. Refer to a Java book if you feel yourself uncomfortable with these satements.

DECISION MAKING – THE IF STATEMENT

```
if (condition) {...} else if (condition) {...} ... else (condition) {...}
```

- About the condition part:
  - Comparison: < > <= >= == !=
  - Double operator is used in logical operations: && ||

LOOPS

```
for( initialStatement; conditionStatement; incrementStatement ) { ... }

while( condition ) { ... }

do { ... } while( condition );

switch / case ...
```

RELATIONS BETWEEN OBJECTS

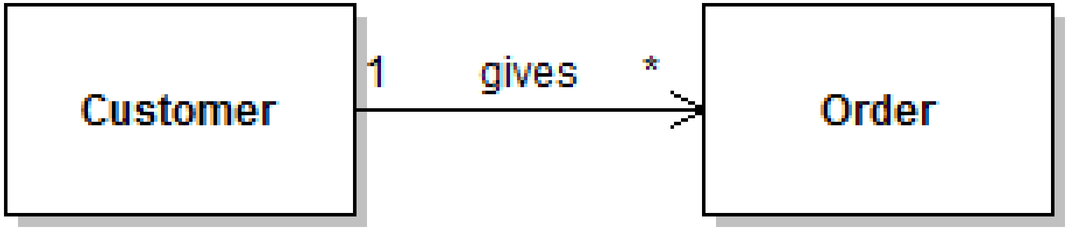
RELATIONS BETWEEN OBJECTS

- We have learned that an object oriented program executes, i.e. runs, by sending messages to objects.
- In order to have an object to send a message to (i.e. use) another object, there must be some kind of relationship between these objects.
- Types of relations:
  - Association
  - Dependency
  - Aggregation
  - Composition
  - Inheritance
- These relations are shown in class diagrams but they should actually be read as relations between instances of classes, i.e. objects.

RELATIONS BETWEEN OBJECTS

ASSOCIATION

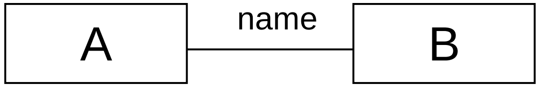
- The essence of association is **ownership**.
- The object that can send a message has the receiver of the message as a member field.
- Example: A customer and his/her orders
  - The logical name and the quantities of the relation are also shown in this diagram.



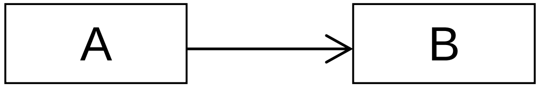
RELATIONS BETWEEN OBJECTS

ASSOCIATION

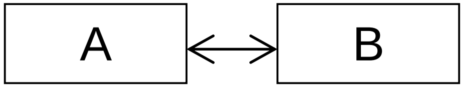
- Representation:



Association



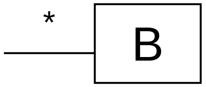
A owns B = instances of A can send messages to instances of B



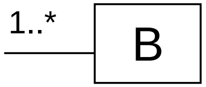
A owns B and B owns A = two-way connection

- The direction of the Arrow is important, it determines who can send messages to whom.
- If no arrows are drawn, this means:
  - either there is a two-way connection,
  - or the direction has not been considered by the architect yet.

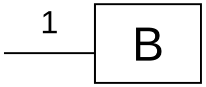
- There may be numbers on the edges of the relation.
  - These numbers represent cardinality,
  - i.e. they show the number of objects at that edge's side.



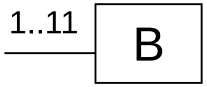
0 or more



1 or more



only 1

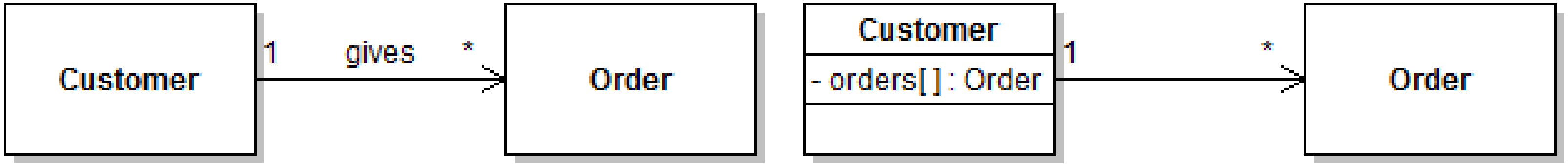


from 1 to 11

RELATIONS BETWEEN OBJECTS

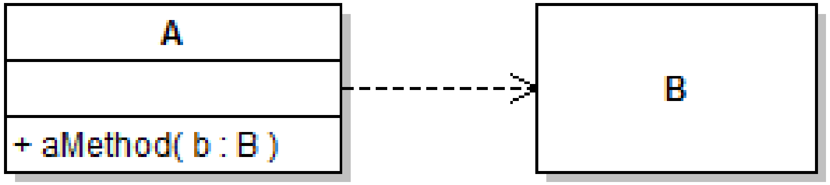
HIDDEN INFORMATION

- If a relation is shown by lines and arrows, you may omit the details within a class.
  - i.e., the two diagrams below are the same.



DEPENDENCY

- The essence of dependency is either **being a parameter of a method** or **temporary usage**, without ownership.
  - Representation:

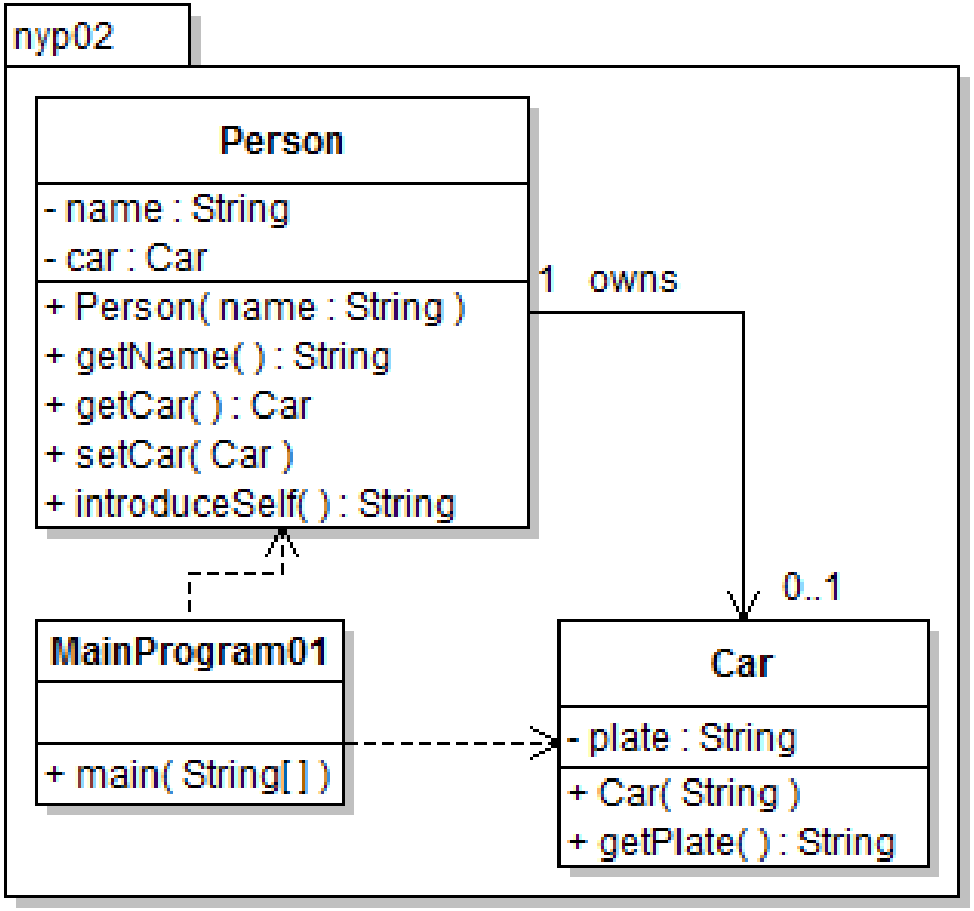


A depends on B = instances of A can send messages to instances of B in the body of aMethod.

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- Let's create a domain model where each person can have a car…
- …and include a program that uses the domain model (must have a main method in order to be run).
- The domain model and the program should reside in different packets.
- Question: Why 0..1 at the association?
- Hidden information: Check the constructor of class Car.



CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- Source code of class Car:

```
package nyp02;

public class Car {
   private String plate;
   public Car( String plateNr ) {
       plate = plateNr;
   }
   public String getPlate() {
       return plate;
   }
}
```

- According to the code, a license plate number is assigned to a car when its created and this number cannot be changed.
- This was easy, lets move on to the class Person:

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- Source code of class Person:

```
package nyp02;
public class Person {
   private String name;
   private Car car;

   public Person( String name ) {
         this.name = name;
   }
   public String getName( ) { return name; }
   public Car getCar( ) { return car; }
   public void setCar( Car car ) { this.car = car; }

   public String introduceSelf( ) {
         String intro;
         intro = "Hello, my name is " + getName();
         if( car != null )
             intro += "and I have a car with license plate "
             + car.getPlate()+ ".";
         return intro;
   }
}
```

Attention!

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- Did you notice in the class diagram that the Car end of the ownership relation between Person and Car is 0..1?
  - This means that not every person may have a car.
- Moreover:
  - When you add a method to a class, there is no guarantee (*) about in what order the methods will run. They may even not be run anyway.
    - (*) except the special rules about constructors and the finalizer.
- As a result, one may create a person but he/she does not have to assign a car to that person.
  - How can one learn the license plate of his/her car when there is not any?
  - In this case, you will encounter with a "NullPointerException" error.
  - Our responsibility is to create solid (without errors and resistant to bugs) code. Therefore:
    - We should check whether a person has a car or not. We should access the license plate of his/her car only if he/she has a car.
      - If a person does not have a car, the value of that member field is **`null`**, meaning that this field is not assigned yet, i.e. it is not initialized.

CODING THE RELATIONS BETWEEN OBJECTS

TESTING FOR INITIALIZATION

- When an object is initialized, we can say that this object is now active.
- We can check whether an object1 is initialized or not as follows:

	Expression	**Value**
Initialized (active)	`object1 == null`	`false`
	`object1 != null`	`true`
Not initialized (inactive)	`object1 == null`	`true`
	`object1 != null`	`false`

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

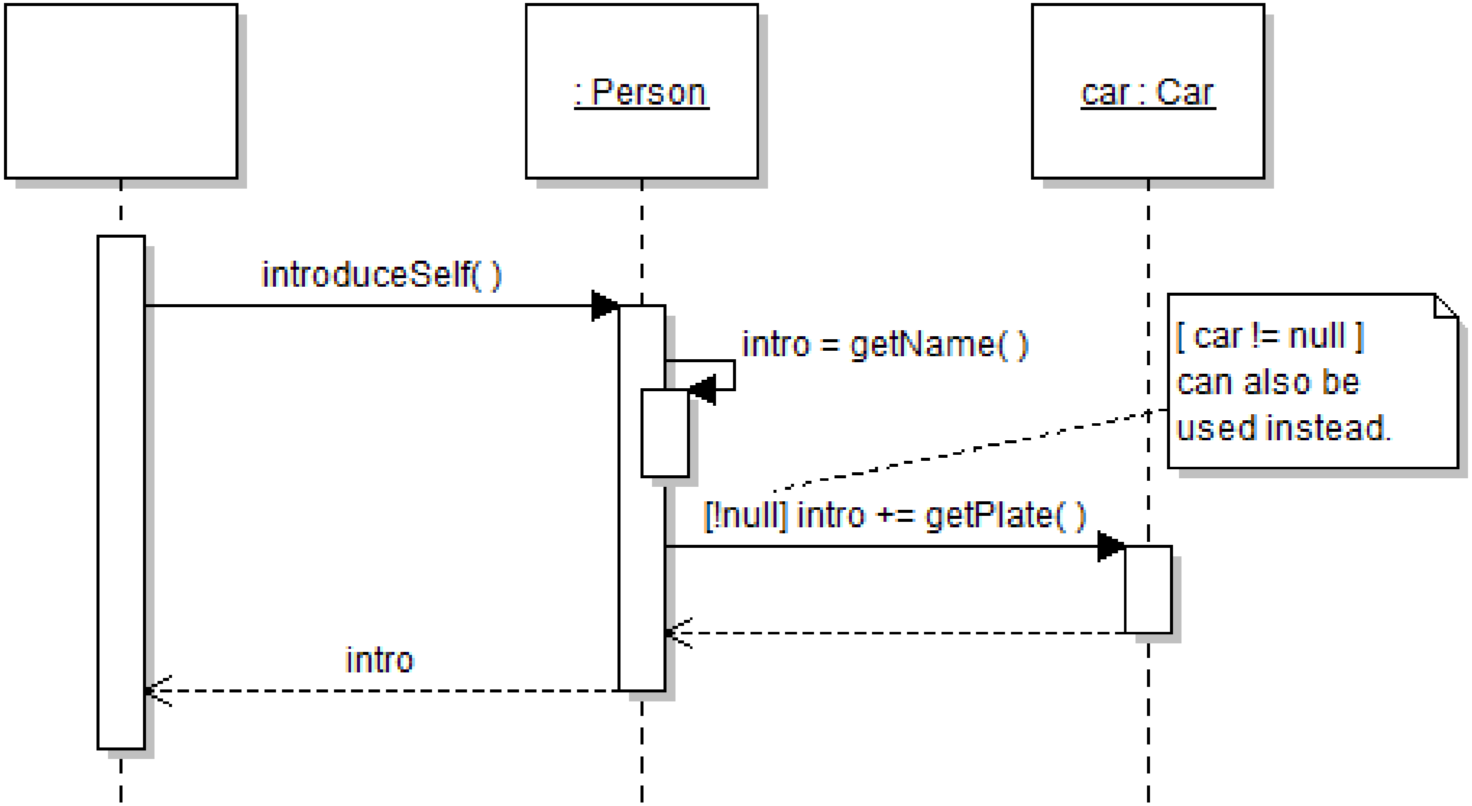
- We can code the MainProgram01 at last:

```
package nyp02;

public class MainProgram01 {
   public static void main(String[] args) {
       Person oktay;
       oktay = new Person( "Oktay Sinanoğlu" );
       Car rover = new Car( "34 OS 1934" );
       oktay.setCar( rover );
       System.out.println( oktay.introduceSelf() );
       Person aziz = new Person( "Aziz Sancar" );
       System.out.println( aziz.introduceSelf() );
   }
}
```

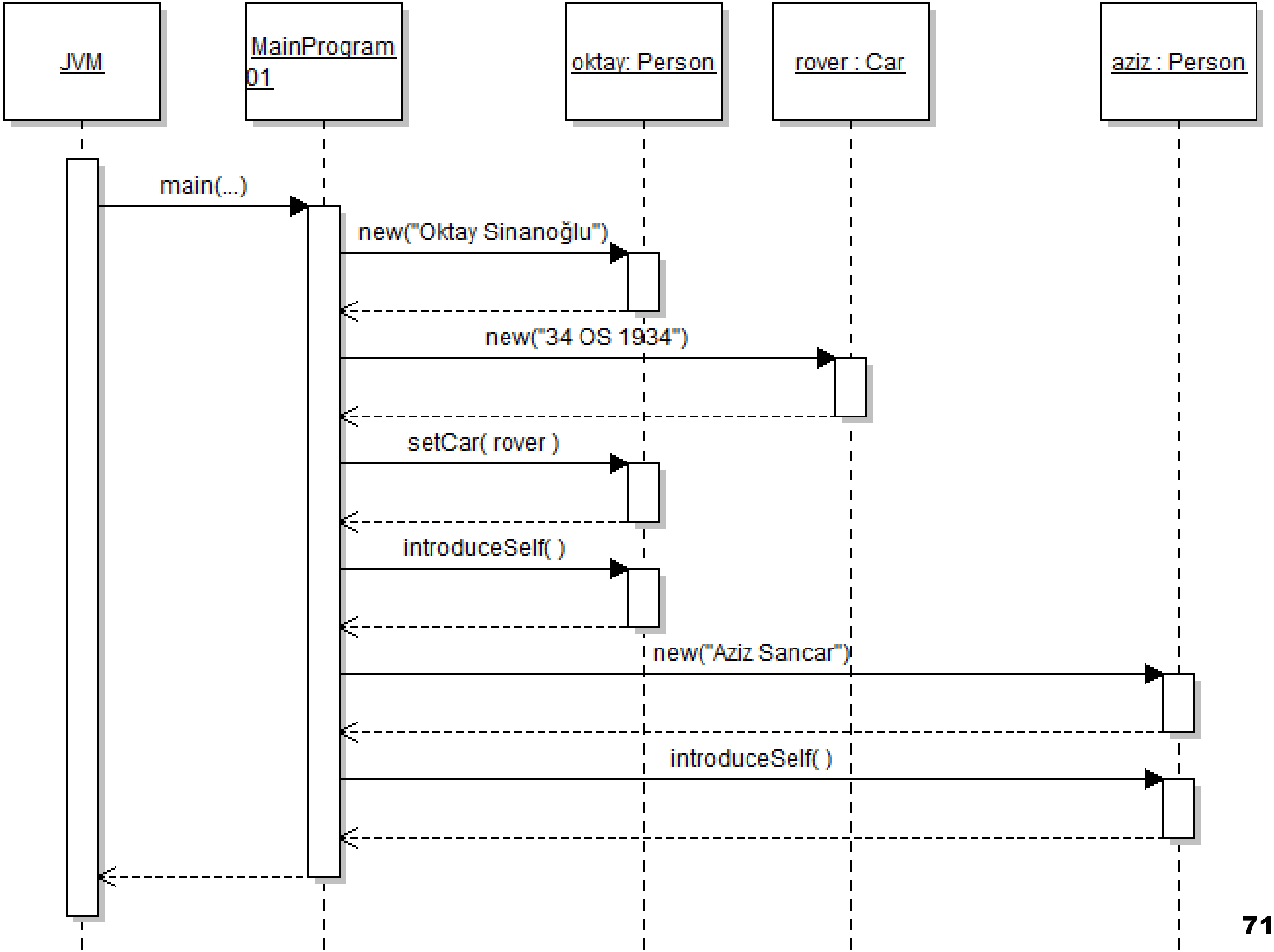
CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : ONE WAY

- Sequence diagram of Person.introduceSelf( ) method:



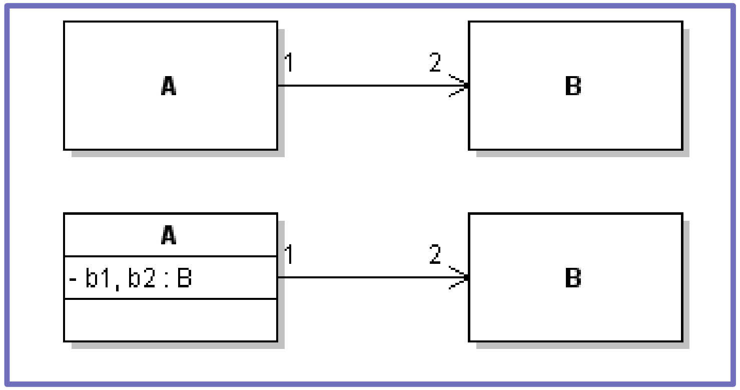
- The sequence diagram for the execution of MainProgram01

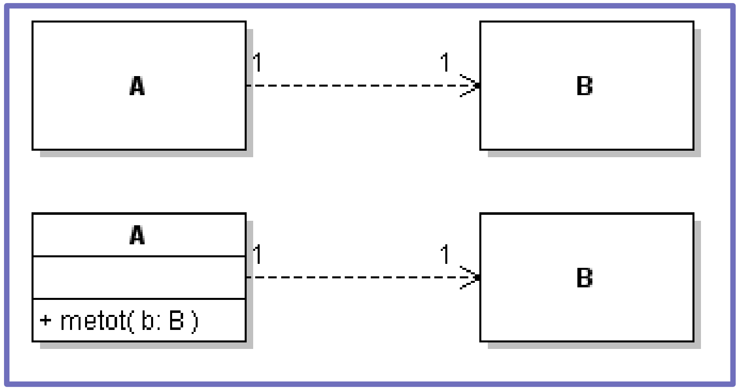


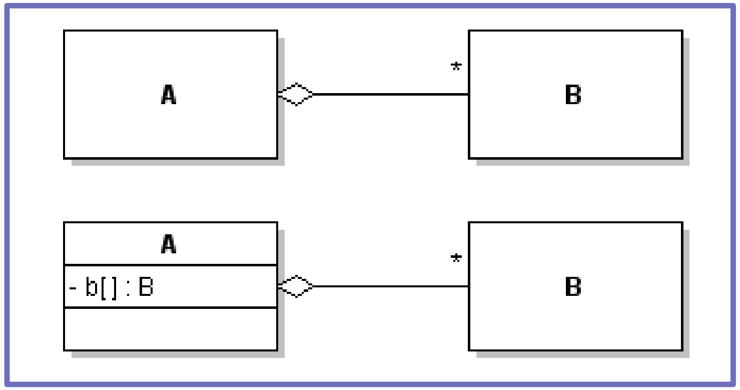
RELATIONS BETWEEN OBJECTS

HIDDEN INFORMATION

- Some implementation details may be hidden in class diagrams.
- In the 3 groups below, the pair on top implies the pair on bottom





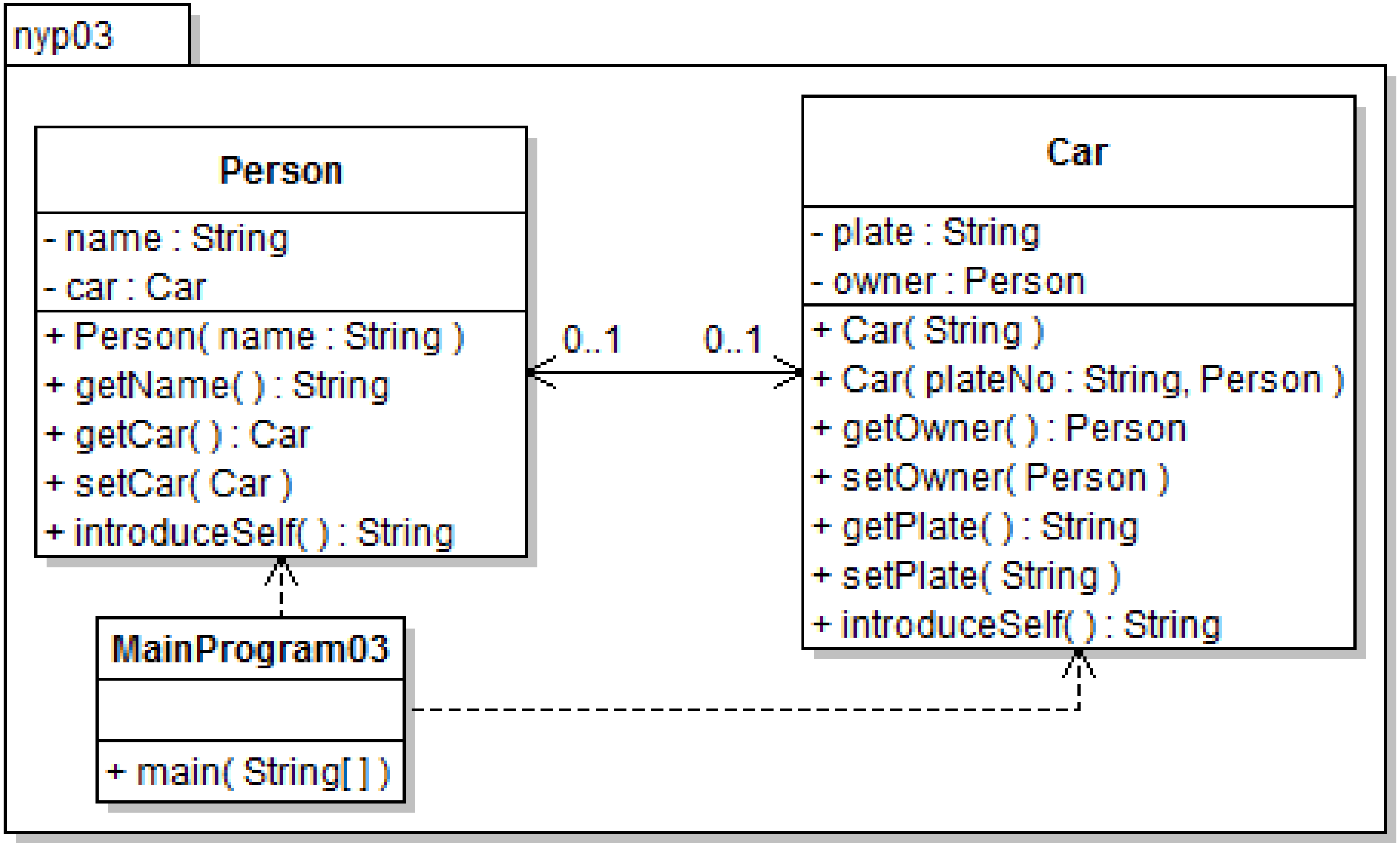


Şekildeki üç ilişki grubunda
üstteki ilişkiler kapalı/gizli,
alttaki ilişkiler açık olarak gösterilmiştir.

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- If we need to be able to find out the owner of a car, as well as being able to assign a car to a person, a two-way association must be constructed.



CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Have you noticed the differences between one-way and two-way associations in the respective class schemas ?
  - We had to change the Car class.
  - The class Person stayed the same.
- We have put the new example into a different package as the Car class needed to be changed.
- About 0..1 on Person side:
  - A car object can be created without an owner in current design (because of Car(plate:String) constructor.
  - Do not be confused: 0..1 on Person side means that zero or more person can be associated with a car, that 0..1 is the quantity of car objects.

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- The source code of the new Car class:

```java
package nyp03;
public class Car {
   private String plate;
   private Person owner;

   public Car( String plate ) { this.plate = plate; }
   public Car( String plate, Person owner ) {
         this.plate = plate;
         this.owner = owner;
   }
   public void setOwner( Person owner ) { this.owner = owner; }
   public Person getOwner() { return owner; }
   public String getPlate( ) { return plate; }
   public void setPlate( String plate ) { this.plate = plate; }
   public String introduceSelf( ) {
         String intro;
         intro = "[CAR] My license plate is " + getPlate();
         if( owner != null )
                 intro += " and my owner is " + owner.getName();
         return intro;
   }
}
```

Attention!

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Why did we have to code the if statement emphasized with the red?
    - Answer: Because one may call the constructor Car(String) and forget to call the setOwner method.
    - Should we remove the Car(String plate) constructor then?
        - No, a car does not have an owner in the real world as soon as it gets out of the factory.

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Let's try what we have done by coding a main method:

```
package nyp03;
public class MainProgram02 {
public static void main(String[] args) {
         Person oktay = new Person("Oktay Sinanoğlu");
         Car rover = new Car("06 OS 1934");
         oktay.setCar(rover);
         rover.setOwner(oktay);
         System.out.println( oktay.introduceSelf() );
         System.out.println( rover.introduceSelf() );

         Person aziz = new Person("Aziz Sancar");
         Car honda = new Car("47 AZ 1946");
         aziz.setCar(honda);
         honda.setOwner(aziz);
         System.out.println( aziz.introduceSelf() );
         System.out.println( honda.introduceSelf() );
   }
}
```

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Can you see a problem in the main method?
  - Why do we have to code both the lines 6 and 7?
  - What if we forget writing any of those lines?
  - What if we mistakenly make a crossover between (oktay, rover) – (aziz, honda)?
  - etc.
- All those defects can be removed by making the two-way association stronger.
  - Which parts of the program should we change?

CODING THE RELATIONS BETWEEN OBJECTS

ASSOCIATION : TWO WAY

- Modifications to classes Person and Car (in a new package):

```
package nyp03b;
public class Person {
        /*the rest is the same*/
        public void setCar( Car car ) {
                this.car = car;
                if( car.getOwner() != this )
                        car.setOwner(this);
        }
}
```

Attention!

```
package nyp03b;
public class Car {
        /*the rest is the same*/
        public void setOwner( Person owner ) {
                this.owner = owner;
                if( owner.getCar() != this )
                        owner.setCar(this);
        }
}
```

Attention!

CODING THE RELATIONS BETWEEN OBJECTS

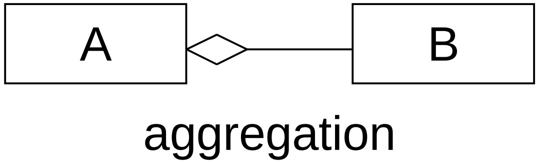
ASSOCIATION : TWO WAY

- How about giving the same flexibility to the second constructor of class Car?
    - Have the car to inform its owner in Car(String,Person)
    - (Instructor does that in class)
- Results:
    - Two-way relations are stronger than one-way relations but they are harder to code.
    - Therefore, if you don't need a two-way relation, code it only one way.
    - What if we need it two-way later?
        - Don't loose time with it, do it later. You will already be busy coding the other requirements.

RELATIONS BETWEEN OBJECTS

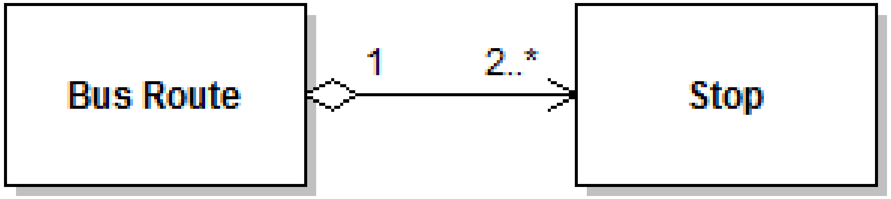
AGGREGATION

- Represents a weak **whole-part relationship**.
- Representation:



- A instances has multiple B instances.
- A: Whole, B: Part.

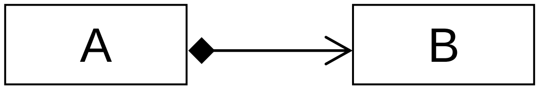
- Even if it is not shown in the diagrams, aggregation implies the following:
  - 1 on the diamond end
  - * (multiplicity) and arrow on the other end
- Agrregation is stronger than association, but only conceptually.
  - It implies that this relation has stronger rules than a regular association.
  - For example, a bus route consists of at least 2 stops and there are rules for adding a new stop to a route.



RELATIONS BETWEEN OBJECTS

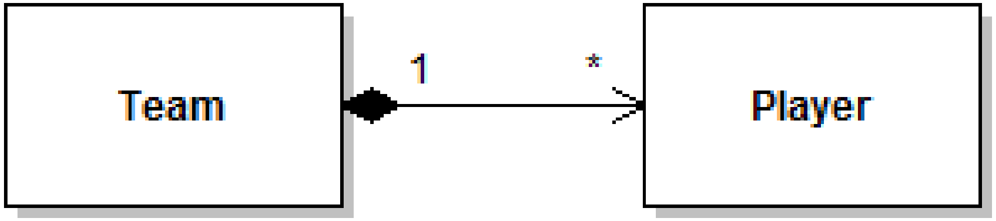
COMPOSITION

- Similar to aggregation, but represents a **stronger whole-part** relation.



Composition

- The strength of composition over aggregation is that in composition, the part can only belong to one whole at the same time.
- Example:



CODING THE RELATIONS BETWEEN OBJECTS

1..* ASSOCIATION, AGGREGATION AND COMPOSITION

- Implementation of 1..* association, aggregation and composition is similar.
- In order to implement multiplicity, we must choose a data structure:
  - Arrays, lists, stacks, queues, heaps, trees, graphs, etc.
  - We will begin with arrays as using object arrays are not much different than using, say, a float array in the C programming language.

CODING THE RELATIONS BETWEEN OBJECTS

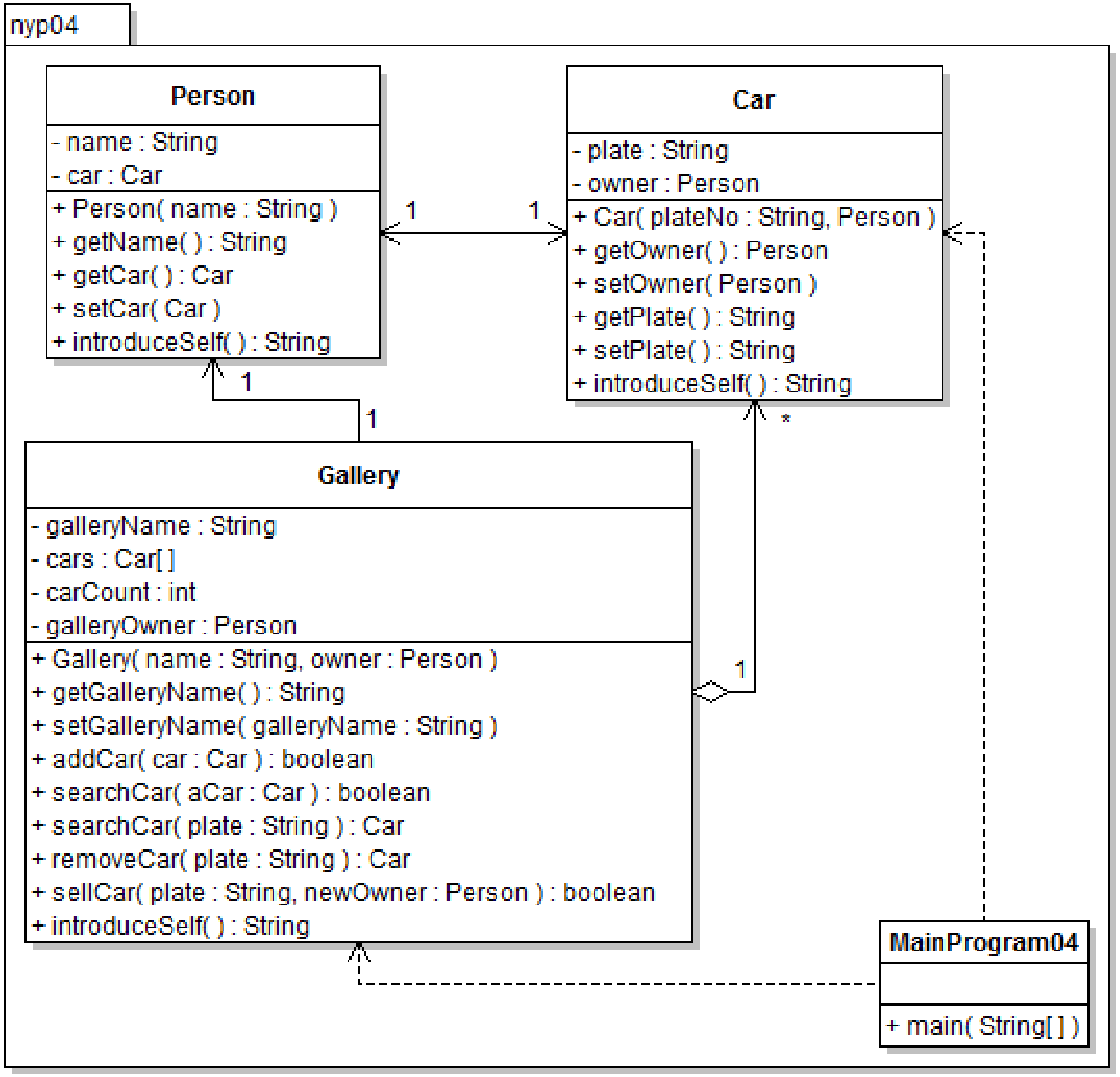
COMPOSITION

- Example: Let’s create a Gallery class associated with multiple cars to be sold.
  - We will reuse the Car class, too.
    - You may want to add a constructor with having only the license plate as parameter. However, I opted for not doing this: We can assign the gallery an owner, a Person instance, as some kind of temporary owner.
  - I opted for representing the relation between the gallery and cars with the composition relation.
    - You can choose to show this relation with 1..* association, it does not matter.
    - This example also demonstrates the arrays and the for loop.

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- UML class diagram:



nyp04

Person

- name : String
- car : Car

+ Person(name : String)
+ getName( ) : String
+ getCar( ) : Car
+ setCar(Car)
+ introduceSelf( ) : String

Car

- plate : String
- owner : Person

+ Car(plateNo : String, Person)
+ getOwner( ) : Person
+ setOwner(Person)
+ getPlate( ) : String
+ setPlate( ) : String
+ introduceSelf( ) : String

Gallery

- galleryName : String
- cars : Car[ ]
- carCount : int
- galleryOwner : Person

+ Gallery(name : String, owner : Person)
+ getGalleryName( ) : String
+ setGalleryName(galleryName : String)
+ addCar(car : Car) : boolean
+ searchCar(aCar : Car) : boolean
+ searchCar(plate : String) : Car
+ removeCar(plate : String) : Car
+ sellCar(plate : String, newOwner : Person) : boolean
+ introduceSelf( ) : String

MainProgram04

+ main(String[ ])

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- Source code of the class Gallery (to be cont'd in the next slides):

```
package nyp04;
public class Gallery {
        private String galleryName;
        private Car[] cars;
        private int carCount;
        private Person galleryOwner;

        public Gallery( String galleryName, Person galleryOwner ) {
            this.galleryName = galleryName; this.galleryOwner = galleryOwner;
            carCount = 0;
            cars = new Car[30];
        }
        public String getGalleryName() { return galleryName; }
        public void setGalleryName(String galleryName) {
           this.galleryName = galleryName;
        }
        public String introduceSelf( ) {
           String intro = "This is a car gallery named "+ galleryName;
           intro += ",  owned by " + galleryOwner.getName();
           intro += ". There are currently " + carCount + " cars to sell.";
           return intro;
        }
```

Note: No constructor is run here, this is just a memory allocation for the array

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- Source code of the class Gallery (cont'd):

```
public boolean addCar( Car aCar ) {
    if(!searchCar(aCar) && carCount < cars.length ){
        cars[ carCount ] = aCar;
        carCount++;
        return true;
    }
    return false;
}//end addCar
//code of class Gallery will continue in
```

You can learn the size of the array this way, but not the element count of the array. Therefore we don't need a variable such as maxCar, but we need a variable such as carCount.
Note:
public final static int maxCar = 30;
private int carCount;

- In the next slides (you can try coding the following by yourself first):
  - It is wise to first check whether the car to be added already exists in the array by calling another method, searchCar.
  - How to sell a car? Implement the sellCar method.
- Excercise/HW: What if you need to take the money issues into account?

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- Source code of the class Gallery (cont’d):

```
public boolean searchCar( Car aCar ) {
    for( Car car : cars )
         if( car == aCar )
             return true;
    return false;
}
public Car searchCar( String plate ) {
    for( int i = 0; i < carCount; i++ )
         if( cars[i].getPlate().compareTo(plate) == 0 )
             return cars[i];
    return null;
}
```

- You can overload the searchCar method, too.
- Source code of the class Gallery will continue in the next slide.

CODING THE RELATIONS BETWEEN OBJECTS

COMPOSITION

- Source code of the class Gallery (cont’d):

```
    public Car removeCar( String plate ) {
        for( int i = 0; i < carCount; i++ ) {
            if( cars[i].getPlate().compareTo(plate) == 0 ) {
                Car theCar = cars[i];
                for( int j = i; j < carCount; j++ )
                    cars[j] = cars[j+1];
                cars[carCount-1] = null; carCount--;
                return theCar;
            }
        }
        return null;
    }
    public boolean sellCar( String plate, Person newOwner ) {
        Car soldCar = removeCar(plate);
        if( soldCar != null ) {
            soldCar.setOwner(newOwner);
            return true;
        }
        return false;
    }
} //end class Gallery
```

CODING THE RELATIONS BETWEEN OBJECTS

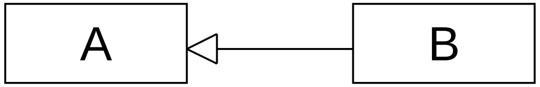
- Source code of the class with main method :

```
package nyp04;
public class MainProgram04 {
    public static void main(String[] args) {
        Gallery cars4U = new Gallery("Cars 4 U",
                 new Person("Yunus Emre Selçuk"));
        Car bmw = new Car("34 RA 440", null);
        Car audi = new Car("06 AC 432",null);
        if( cars4U.addCar(bmw) ) {
            System.out.println("Adding operation succeded");
        }
        else { System.out.println("Adding operation failed"); }
        cars4U.addCar(audi);
        System.out.println(cars4U.introduceSelf());
        System.out.println(bmw.introduceSelf());
        if( cars4U.searchCar(bmw) == true ) {
            System.out.println("Search is successful");
        }
        else     System.out.println("Search has failed");
        if( cars4U.removeCar(audi.getPlate()) == audi )
            System.out.println("Remove operation succeeded");
        else     System.out.println("Remove operation failed");
        if( cars4U.searchCar(audi) == false )
            System.out.println("Last operation has correctly failed");
        else     System.out.println("Last operation has incorrectly succeeded");
    }
}
```

RELATIONS BETWEEN OBJECTS

INHERITANCE

- Real world: A child inherits genetic properties from his/her parents.
- OOP: A means of creating new classes from an existing class, in a way that is similar with the real world.
- Representation:



Inheritance

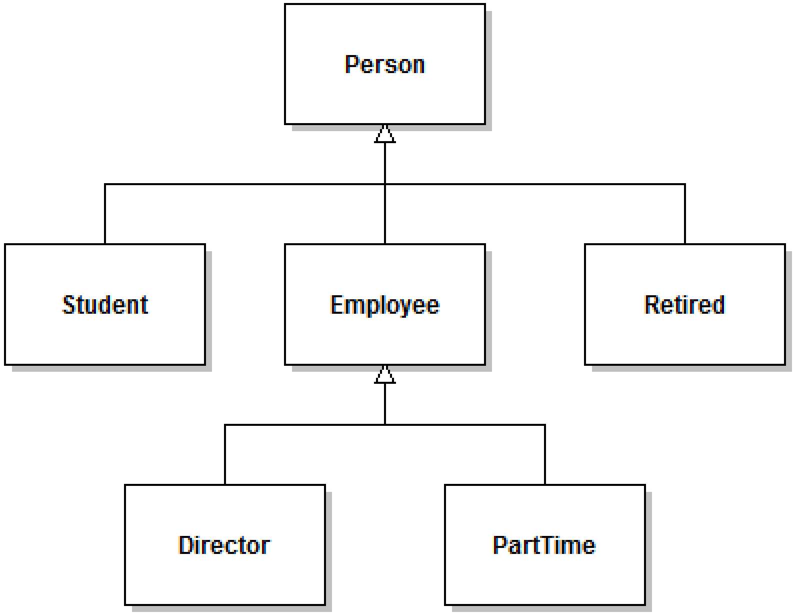
- Pay attention to the direction of the arrow!

- A:
  - Parent class
  - Super class
  - Base class
- B:
  - Child class
  - Sub class
  - Derived class
- How inheritance works:
  - All member fields and methods of the parent are transferred to the child
    - However, children cannot access the inherited private members
  - Protected members and inheritance:
    - Those members can be accessed by children but they are inaccessible for other classes

RELATIONS BETWEEN OBJECTS

INHERITANCE

- Rules of the inheritance mechanism:
  - Sub classes cannot reject a member from the super class.
  - However, bodies of inherited methods can be changed
    - This is called **overriding**.
    - Attention: **final methods cannot be overridden.**
  - New members can be added to sub classes.
  - A sub class can be the parent of other classes. The tree structure created this way is called as inheritance hierarchy or as inheritance tree.

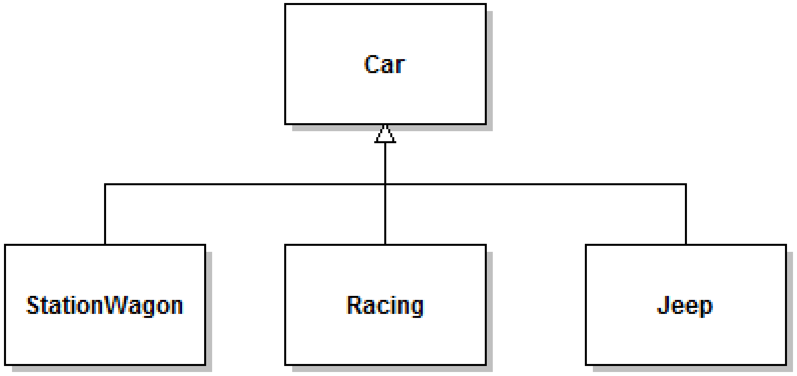


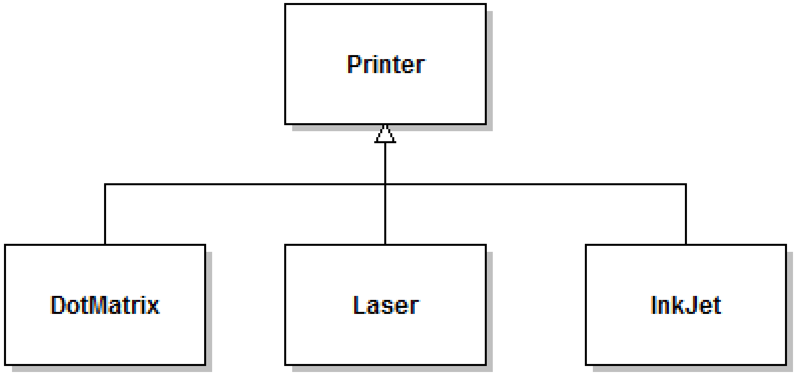
- Do not make the tree too deep
  - It leads to the fragile super class problem.
  - When you change the upmost super class, this change will affect all other sub classes.
  - This is similar to a tree that has a rotten root, it can easily collapse with the wind.

RELATIONS BETWEEN OBJECTS

INHERITANCE

- Effects of inheritance
  - Inheritance is also called as the generalization – specialization relation
    - The child is a specialized and more capable version of its parent.
    - Likewise, the parent is a generalized version of its children with less capabilities
  - Substitutability:
    - The child can be used wherever the parent is expected.
    - Therefore inheritance is also called as IS-A relationship.

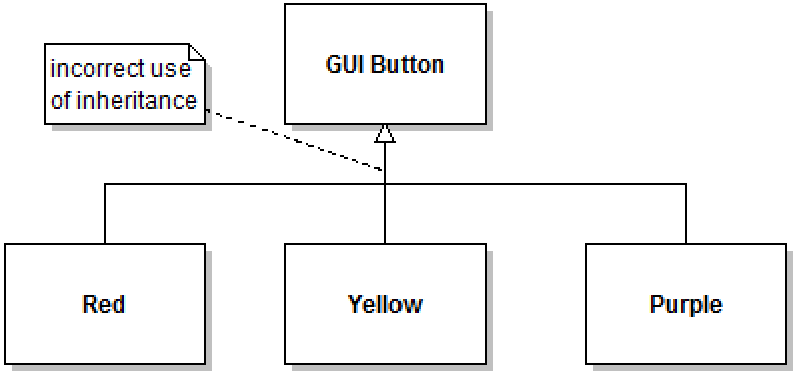


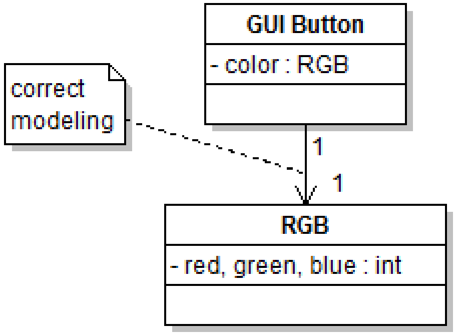


RELATIONS BETWEEN OBJECTS

INHERITANCE

- Misuse of inheritance

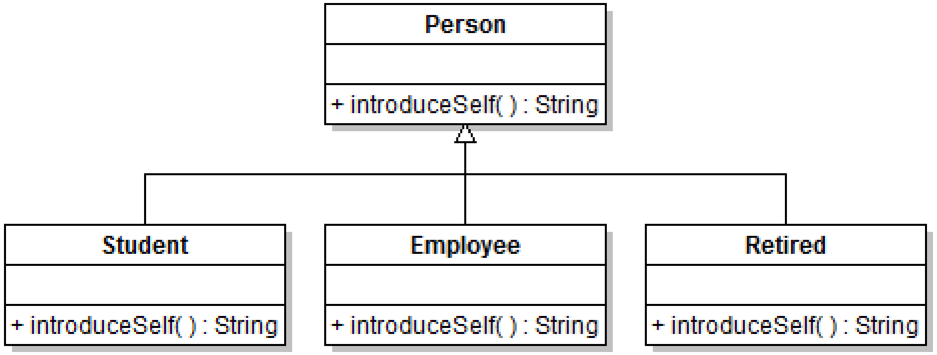




SPECIAL TOPICS in INHERITANCE

POLYMORPHISM and OVERRIDING

- We have learned that the body of an inherited method can be changed, and this act is named overriding.
  - Remember: final methods cannot be overridden.
- We have also learned that an instance of a subclass can be used wherever an instance of its superclass is expected.


Person
+ introduceSelf( ) : String
Student
+ introduceSelf( ) : String
Employee
+ introduceSelf( ) : String
Retired
+ introduceSelf( ) : String

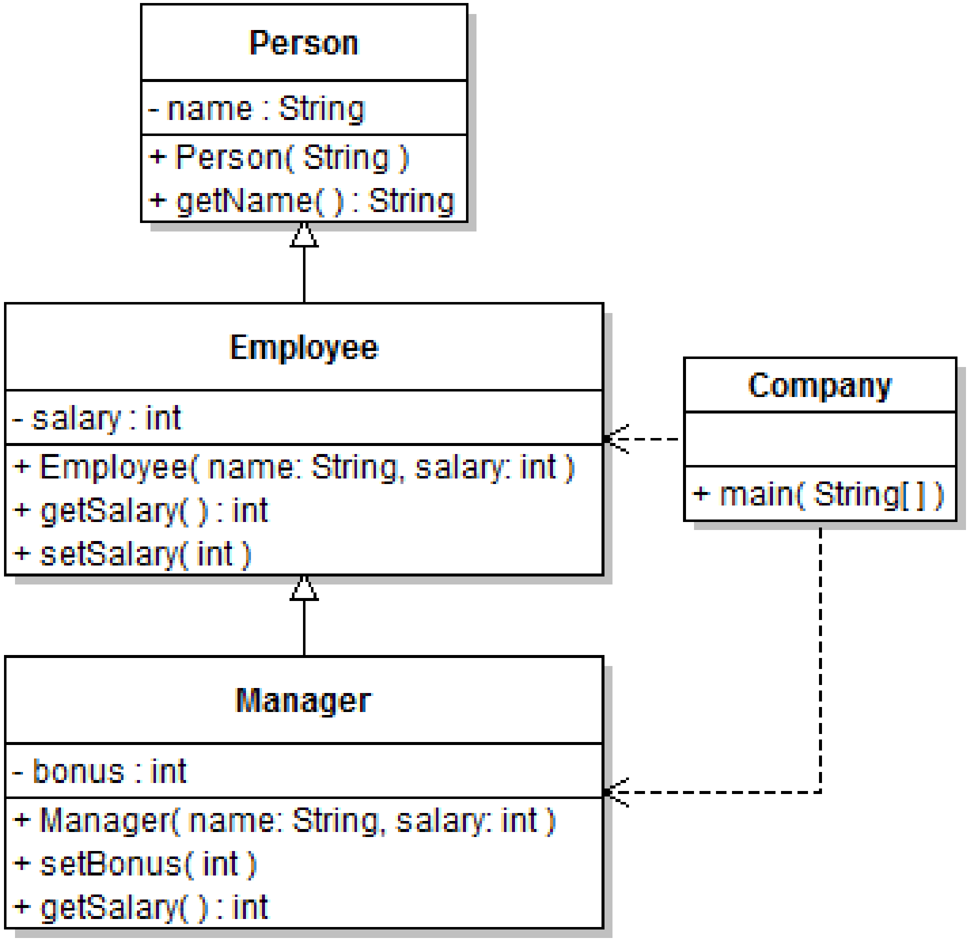
- An example inheritance tree is shown on the left
  - The introduceSelf() method is overridden in the subclasses.

- Consider an array of type Person, having mixed instances of all classes above. What happens if we run the introduceSelf method of all members in the array?
  - The correct version of the introduceSelf method is executed in runtime.
  - This mechanism is called **polymorphism**.
- In this case, what should we do if we need to access an overridden method's previous version, i.e. as it is coded in the super class?
  - We can access that particular method via the **`super`** pointer!

SPECIAL TOPICS in INHERITANCE

POLYMORPHISM and OVERRIDING

- UML class schema of an example which consists of:
  - An inheritance tree (Person – Employee – Manager),
  - And a class using them (Company)



SPECIAL TOPICS in INHERITANCE

POLYMORPHISM and OVERRIDING

- Source codes:

```
package nyp05;
public class Person {
        private String name;
        public Person( String name ) { this.name = name; }
        public String getName( ) { return name; }
}
```

```
package nyp05;
public class Employee extends Person {
        private int salary;
        public Employee( String name, int salary ) {
                super( name );
                this.salary = salary;
        }
        public int getSalary( ) { return salary; }
        public void setSalary( int salary ) { this.salary = salary; }
}
```

Mention the important role of super

SPECIAL TOPICS in INHERITANCE

POLYMORPHISM and OVERRIDING

- Source codes (continued):

```
package nyp05;

public class Manager extends Employee {
    private int bonus;

    public Manager( String name, int salary ) {
        super( name, salary );
        bonus = 0;
    }
    public void setBonus( int bonus ) {
        this.bonus = bonus;
    }
    public int getSalary( ) {
        return super.getSalary( ) + bonus;
    }
}
```

Don't cause an error by writing:
`super(name)`
`super(salary)`

Remember visibility rules.
Cannot simply write:
`salary + bonus`

- Attention: You can use **`super`** only once. You cannot write **`super.super`**
- Attention: The **`super`** reference must be the first statement in the constructor..

SPECIAL TOPICS in INHERITANCE

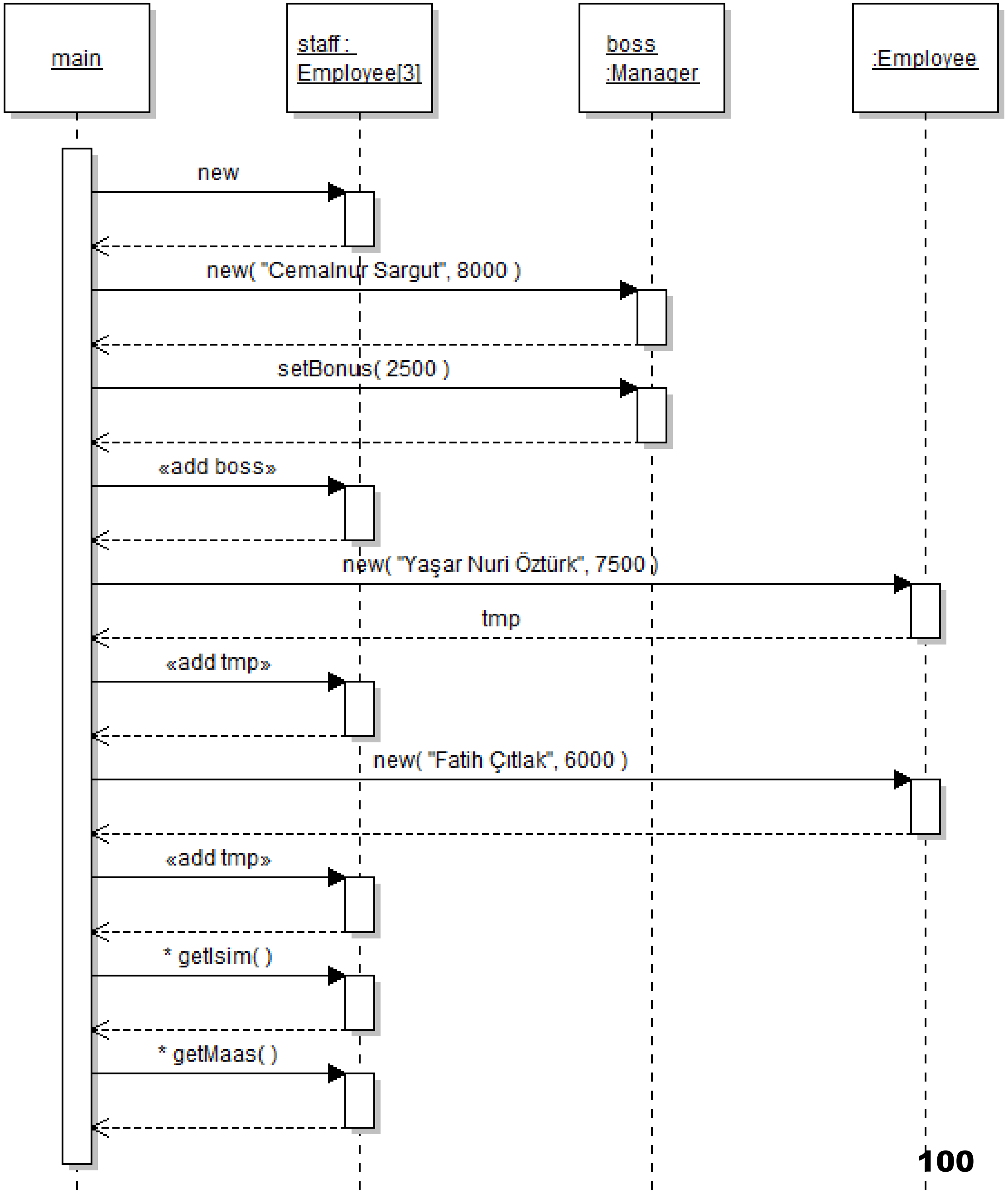
POLYMORPHISM and OVERRIDING

- Source codes (continued):

```
package nyp05;
public class Company {
    public static void main(String[] args) {
       Employee[] staff = new Employee[3];
       Manager boss = new Manager( "Cemalnur Sargut", 8000 );
       boss.setBonus( 2500 );
       staff[0] = boss;
       staff[1] = new Employee( "Yaşar Nuri Öztürk", 7500 );
       staff[2] = new Employee( "Fatih Çıtlak", 7000 );
       for( Employee author : staff )
           System.out.println( author.getName() + " " +
             author.getSalary( ) );
   }
}
```

- Have you noticed the syntax of the for loop?

- Sequence diagram of the main method:



SPECIAL TOPICS in OOP

OVERLOADING and OVERRIDING

- Do not confuse overriding and overloading:
  - Override: Modifying the body of inherited method. Overriding is closely related with inheritance.
  - Overload: Have multiple methods having same names but with different parameters. Overloading is not related with inheritance.
- Overriding example: The getSalary method in class Manager.
- Overloading example: Let's overload the constructor of the class Manager by adding the following constructor method:

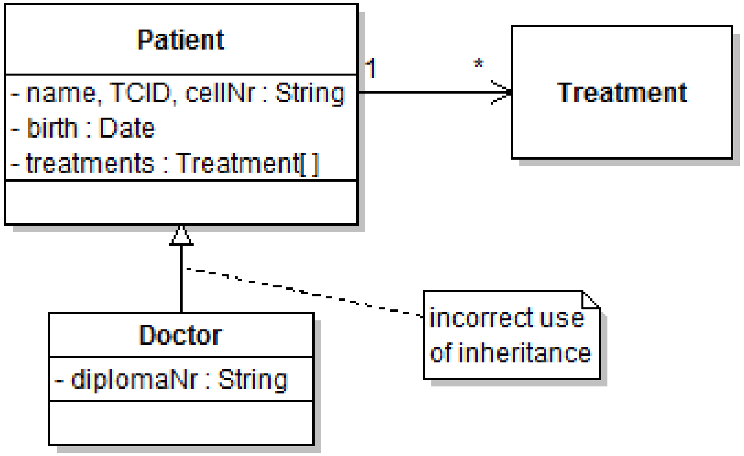
```
public Manager( String name, int salary, int bonus ) {
       super( name, salary );
       this.bonus = bonus;
}
```

  - Now the class Manager has two constructors.

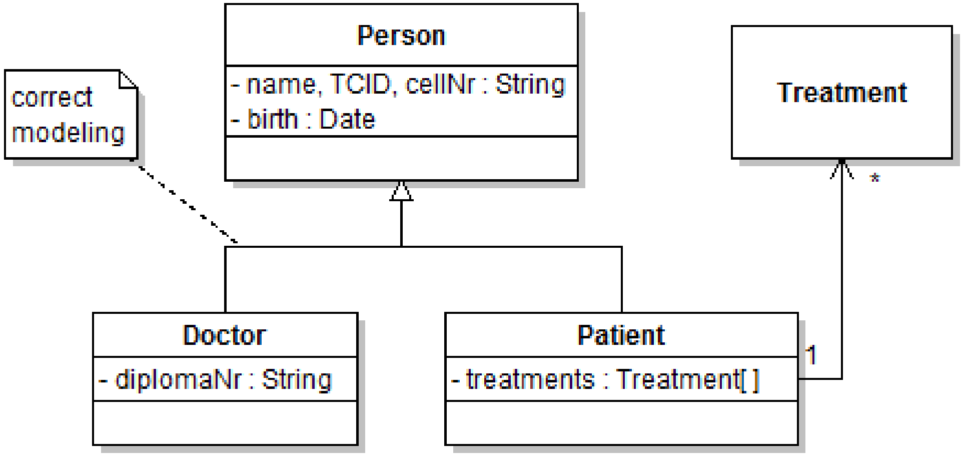
RELATIONS BETWEEN OBJECTS

INHERITANCE

- An excerpt from a requirements documentation:
  - We should keep track of the patients' names, TC ID numbers, birth dates and cell numbers. This information should be stored for doctors, too.In addition, it is required by law to keep the diploma numbers of doctors. We should keep records of treatments applied to a patient, too.
- Wrong modeling: Correct modeling:



Patient
- name, TCID, cellNr : String
- birth : Date
- treatments : Treatment[ ]
1
*
Treatment
Doctor
- diplomaNr : String
incorrect use of inheritance



- Patients should have treatments, not the doctors. In the incorrect case, the treatments array is transferred to doctors because of the inheritance relation.

CODING THE RELATIONS BETWEEN OBJECTS

INHERITANCE AND THE COSMIC SUPER CLASS IN JAVA

- In Java, the class java.lang.Object is the implicit super class of all classes.
- You can override some methods of this class in your classes for your own purposes:
  - public String toString(): You can return a string representation of the object that is easy for a person to read.
    - Just as you have done in `public String Car.introduceSelf()`
    - The advantage of overriding toString is that you can print the instance directly.

SPECIAL TOPICS in OOP

ABSTRACT CLASSES

- An abstract class is such a class that it is used as a base class and it represents a template for its regular sub classes.
  - Regular classes we have coded so far can be called concrete.
  - If a class is abstract, we identify it with the keyword **`abstract`**.
- It is forbidden to create instances of an abstract class.
- One can create instances of concrete subclasses of an abstract class.
- Abstract classes can have member fields, just like the concrete classes.
- Abstract classes can have both concrete and abstract member methods.
  - An abstract method has only definiton together with the keyword **`abstract`**, it does not have a body.
- The bodies of inherited abstract methods must be defined in the concrete subclasses.
  - Otherwise, those subclasses should also be defined as abstract.

SPECIAL TOPICS in OOP

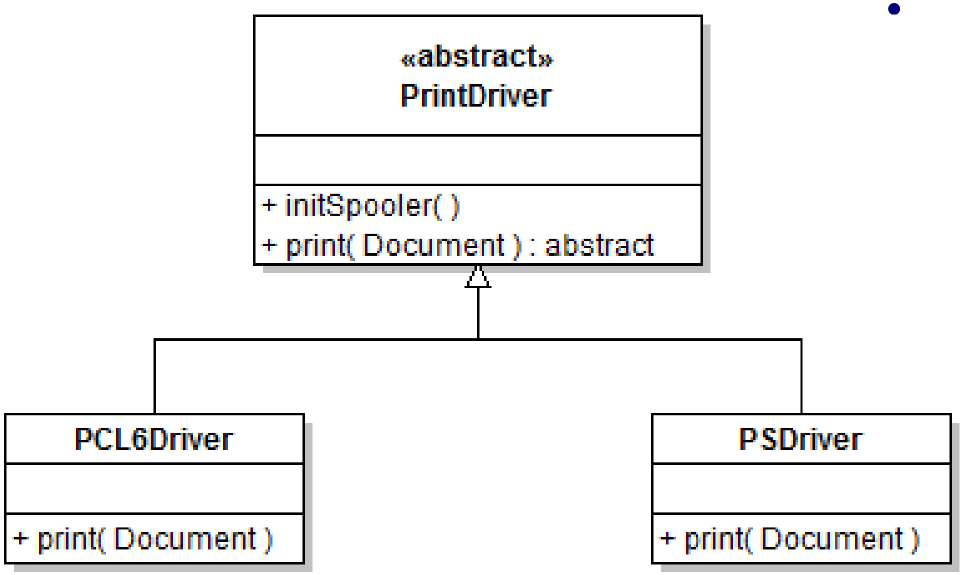
ABSTRACT CLASSES

- When do we need abstract classes?
  - The more we climb upwards in a class hierarchy, the more the classes become generalized. At a point, the classes may become so generalized that we don't need them to be instantiated.
  - We said that you may use an abstract class as a template. In this case:
    - If you need to be make sure that a particular class must have some particular methods, you can define these methods in an abstract super class and you introduce an inheritance relationship between the aforementioned classes.
- You can mark the abstract classes in UML class schemas in italics or by adding the <<Abstract>> stereotype.
  - <<…>>: This is called a stereotype and used in any kind of UML schema whenever a symbol is used without its regular meaning.

SPECIAL TOPICS in OOP

ABSTRACT CLASSES

- Example:



- In the super class, we know how to initialize the spooler but we don't know the exact details of the printing process. We only know that a printer can print documents.
  - Therefore, PrintDriver instances are of no use to us.
  - However, the PrintDriver class removes the burden of coding the spooler initialization method from the coders of the subclasses.

- The details of the printing process are coded in the sub classes.
- Our design allows to install multiple printers of types PCL6, PS and printers of any other future types can be installed at the same computer. All those different printer types can be accessed in a uniform fashion represented by the PrintDriver class.

SPECIAL TOPICS in OOP

ABSTRACT CLASSES

- Source codes of the example:

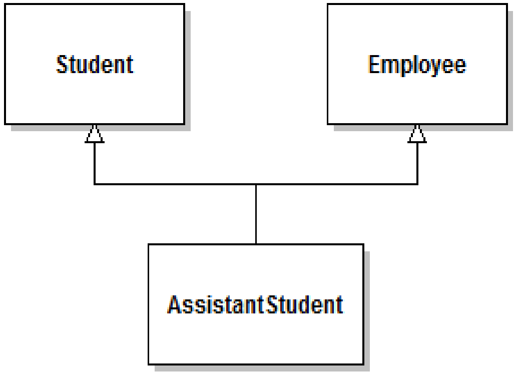
```
public abstract class PrintDriver {
    public void initSpooler( ) {
       /* necessary codes*/
    }
    public abstract void print( Document doc );
}
```

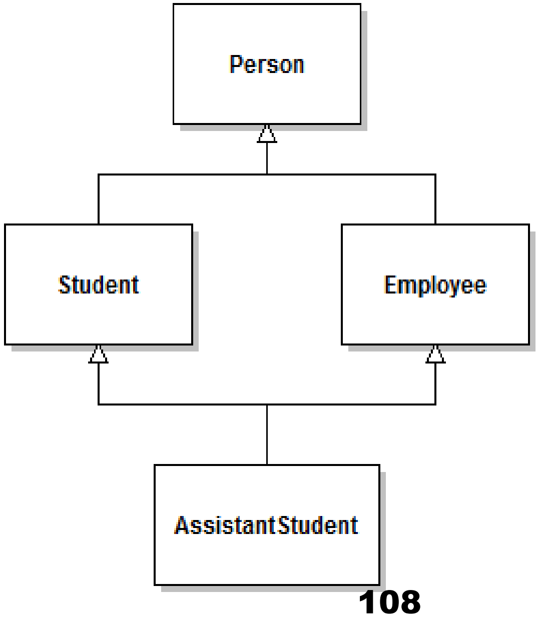
```
public class PCL6Driver extends PrintDriver {
    public void print(Document doc) {
       //necessary code is inserted here
    }
}
```

SPECIAL TOPICS in OOP

MULTIPLE INHERITANCE

- If a sub class can have more than one super class, this is called multiple inheritance.
  - For example, we have classes Student and Employee.
  - Our client asked for adding an AssistantStudent type to the software
  - At the first sight, it can be convenient to create this new class via inheriting from existing classes.
- However, multiple inheritance comes with an added complexity, i.e. the Diamond Inheritance Problem:
  - Consider the toString method, overriden in classes Student and Employee.
  - Which version of the toString method will be inherited by our new class?
  - In this case, we absolutely have to override the toString method in the new class.
- Due to such added complexities, multiple inheritance is not supported in recent languages such as Java and C#

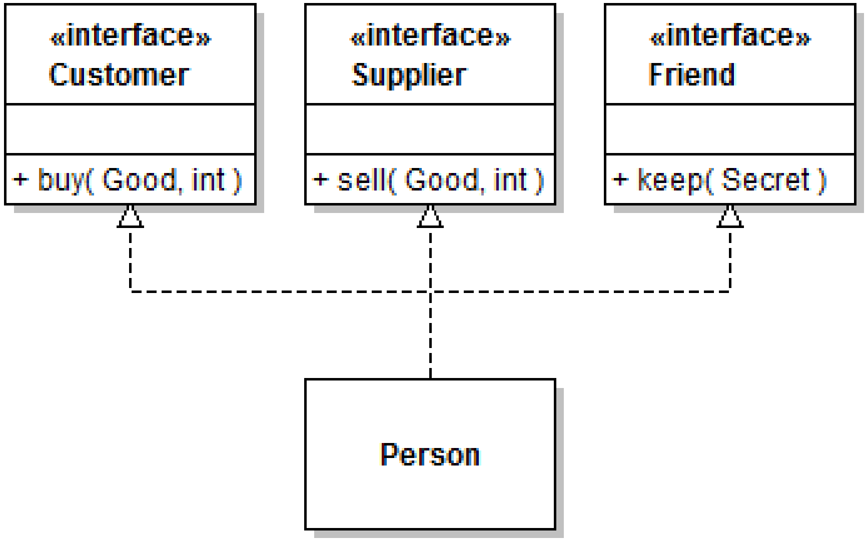




SPECIAL TOPICS in OOP

INTERFACES

- Interfaces can be thought as abstract classes without members.
  - If you wish, you may add "public final static" member fields only.
- An interface is a named collection of methods.
- UML representation and source code of an example:



```
public interface Customer {
    public void buy( Good aGood, int quantity );
}
public interface Supplier {
    public void sell( Good aGood, int quantity );
}
public interface Friend {
    public void keep( Secret aSecret );
}
public class Person implements Customer,
                  Supplier, Friend {
    public void buy( Good aGood, int quantity ) {
                  //related code
    }
    public void sell (Good aGood, int quantity ) {
                  // related code
    }
   public void keep( Secret aSecret ) {
                  // related code
    }
}
```

SPECIAL TOPICS in OOP

INTERFACES

- We use interfaces …
  - in order to group responsibilites of entities,
  - in order to give objects multiple views,
  - instead of inheritance,
    - Because inheritance is a "heavy weight" relation that should be used only when it is absolutely necessary.
  - instead of multiple inheritance.

SPECIAL TOPICS in OOP

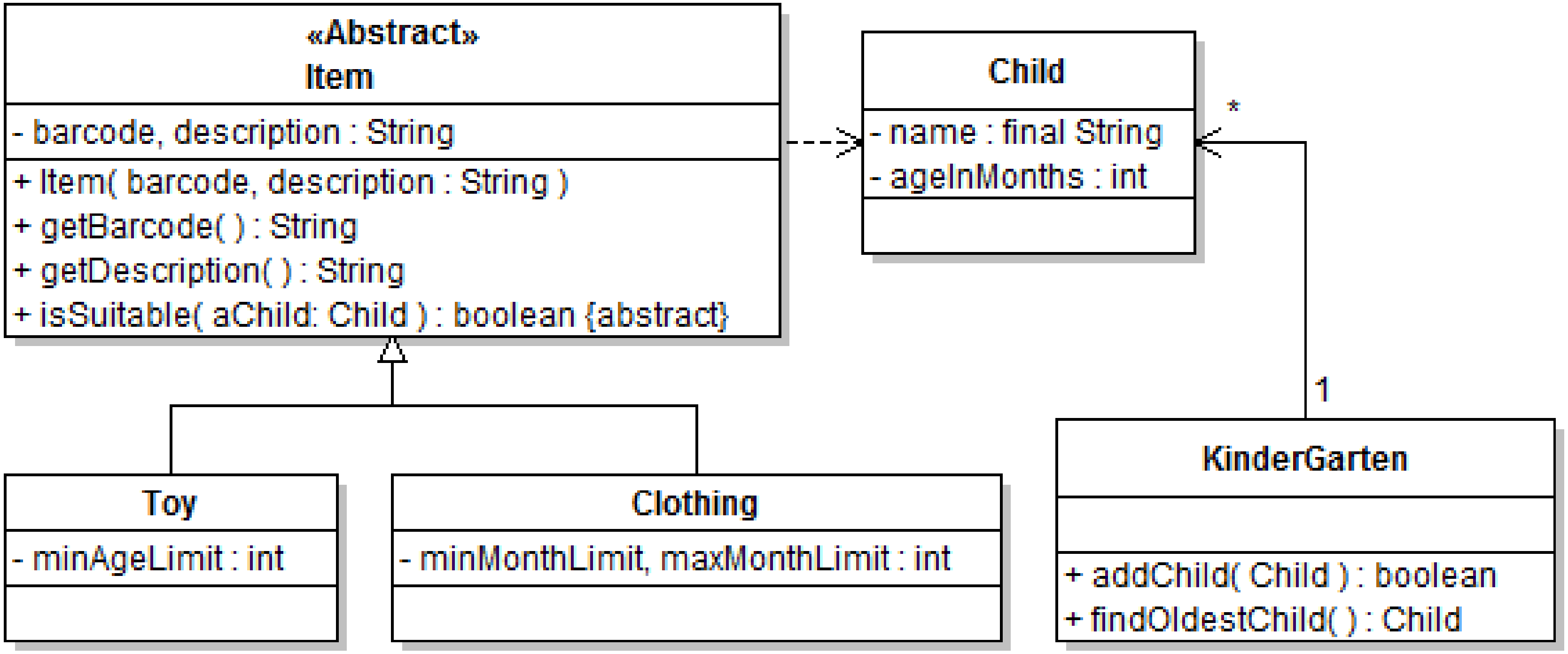
INTERFACES

- Rules related to interfaces:
  - A class should code the bodies of all the methods of the implemented interfaces.
  - Regular member fields cannot be defined in interfaces. Interfaces can only have "public final static" member fields.
  - Only public methods can be defined in interfaces.
  - Interfaces cannot have constructors.
  - A class can implement multiple interfaces.
  - I suggest you to begin naming interfaces with I (capital i).

SPECIAL TOPICS in OOP

DESIGNING AND CODING AN ABSTRACT CLASS

- Consider items for children:
  - Not every item is suitable for every child.
    - Toys have lower age limit, usually measured in years.
    - Clothes have both lower and higher age limits, usually measured in months.
- How should we model this case?



«Abstract»
Item
- barcode, description : String
+ Item(barcode, description : String)
+ getBarcode( ) : String
+ getDescription( ) : String
+ isSuitable(aChild: Child) : boolean {abstract}
Child
- name : final String
- ageInMonths : int
*
1
Toy
- minAgeLimit : int
Clothing
- minMonthLimit, maxMonthLimit : int
KinderGarten
+ addChild(Child) : boolean
+ findOldestChild( ) : Child

DESIGNING AND CODING AN ABSTRACT CLASS

- Source code of class Item:

```
package nyp06;
public abstract class Item {
    private String barcode, description;
    public Item(String barcode, String description) {
        this.barcode = barcode;
        this.description = description;
    }
    public String getBarcode() {
        return barcode;
    }
    public String getDescription() {
        return description;
    }
    public abstract boolean isSuitable(Child aChild);
}
```

- The logic for determining the suitability of an Item is different for a Toy and a Clothing. Therefore we have left the isSuitable method as abstract here.
- However, we have coded the common operations in the abstract base class so that we don't have to code them again in sub classes.
    - "Say a word only once and at the right time! "

SPECIAL TOPICS in OOP

DESIGNING AND CODING AN ABSTRACT CLASS

- Source code of concrete subclasses:

```
package nyp06;
public class Clothing extends Item {
    private int minMonthLimit, maxMonthLimit;

    public Clothing(String barcode, String description,
                int minMonthLimit, int maxMonthLimit ) {
        super(barcode, description);
        this.minMonthLimit = minMonthLimit;
        this.maxMonthLimit = maxMonthLimit;
    }
    public boolean isSuitable(Child aChild) {
        if( aChild.getAgeInMonths() >= minMonthLimit
                && aChild.getAgeInMonths() <= maxMonthLimit )
            return true;
        return false;
    }
}
```

SPECIAL TOPICS in OOP

DESIGNING AND CODING AN ABSTRACT CLASS

- Source code of concrete subclasses:

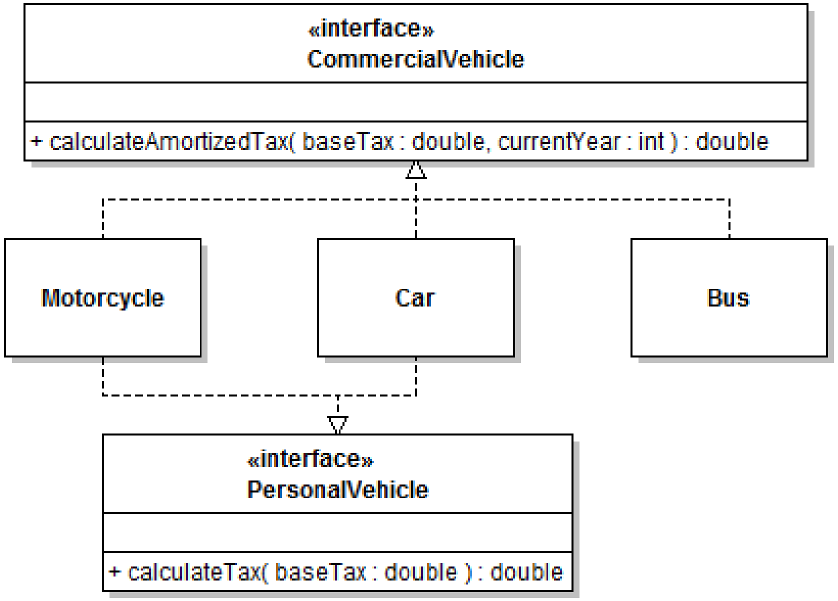
```java
package nyp06;
public class Toy extends Item {
    private int minAgeLimit;

    public Toy(String barcode, String description, int minAgeLimit) {
        super(barcode, description);
        this.minAgeLimit = minAgeLimit;
    }
    public boolean isSuitable(Child aChild) {
        if( aChild.getAgeInMonths()/12 >= minAgeLimit )
            return true;
        return false;
    }
}
```

- You can implement the class Kindergarten with the given methods and more as exercise
- You can build different relationships between Item instances at one end and Kindergarten/Child at the other end(s)

SPECIAL TOPICS in OOP

DESIGNING AND CODING INTERFACES

- Consider the following requirement about calculating the taxes of vehicles:
  - Taxation of commercial and personal vehicles is different.
  - Motorcycles, cars and buses can be registered as commercial vehicles.
  - Only motorcycles and cars can be registered as personal vehicles .
  - Only taxes of commercial vehicles can be amortized.
  - Commercial or not, calculation of the tax of different vehicles (car, bus, etc) are very different.
- How can we model this requirement?



«interface»
CommercialVehicle
+ calculateAmortizedTax(baseTax : double, currentYear : int) : double
Motorcycle
Car
Bus
«interface»
PersonalVehicle
+ calculateTax(baseTax : double) : double

- Hint: If the tax calculation for different vehicles were similar (i.e. parametrized), using one abstract base class instead of interfaces would be a better choice.

SPECIAL TOPICS in OOP

DESIGNING AND CODING INTERFACES

- Coding the interfaces:

```java
package nyp07;
public interface CommercialVehicle {
    public double calculateAmortizedTax( double baseTax, int currentYear );
}
```

```java
package nyp07;
public interface PersonalVehicle {
    public double calculateTax( double baseTax );
}
```

SPECIAL TOPICS in OOP

DESIGNING AND CODING INTERFACES

- Coding the class car:

```
package nyp07;
public class Car implements CommercialVehicle, PersonalVehicle {
    private String plate;
    private int modelYear;
    private double engineVolume;
    public Car(String plate, int modelYear, double engineVolume) {
        this.plate = plate; this.modelYear = modelYear;
        this.engineVolume = engineVolume;
    }
    public double calculateTax( double baseTax ) {
        return baseTax * engineVolume;
    }
    public double calculateAmortizedTax( double baseTax, int currentYear ) {
        //Tax can be reduced %10 for each year as amortization
        int age = currentYear - modelYear;
        if( age < 10 )
            return baseTax * engineVolume * (1-age*0.10);
        return baseTax * engineVolume * 0.10;
    }
    public String getPlate() { return plate; }
    public int getModelYear() { return modelYear; }
    public double getEngineVolume() { return engineVolume; }
}
```

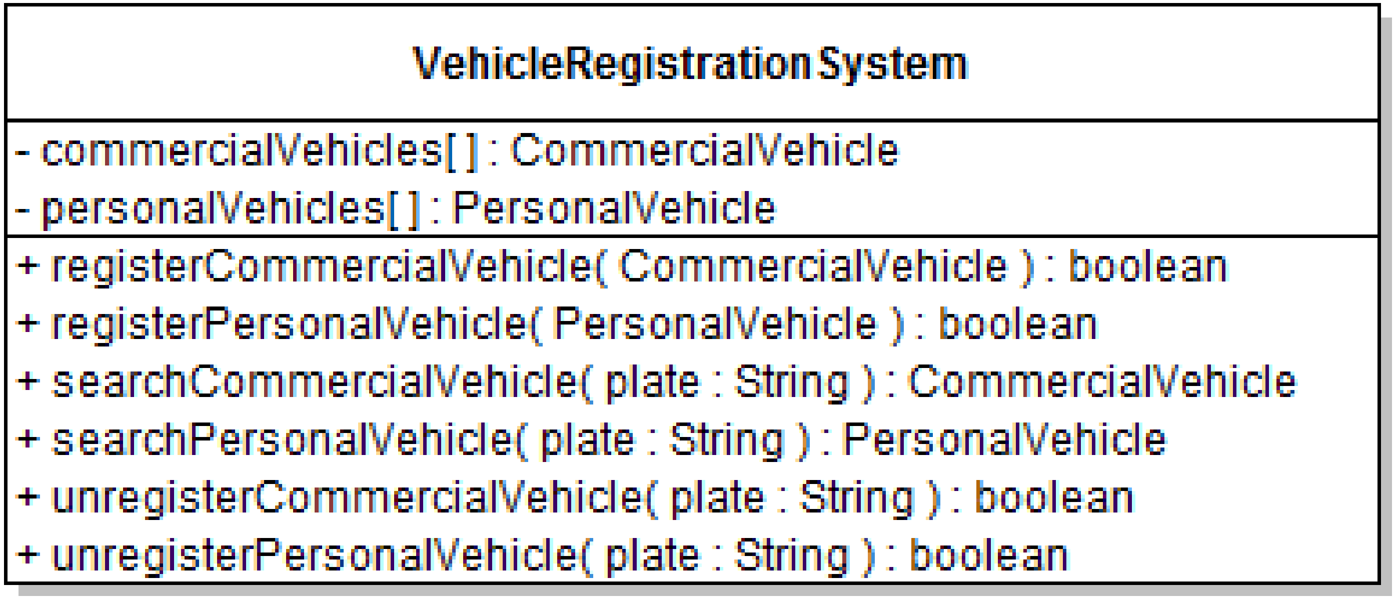
DESIGNING AND CODING INTERFACES

- Coding the class bus:

```
package nyp07;
public class Bus implements CommercialVehicle {
    private String plate;
    private int modelYear;
    private double tonnage;
    public Bus(String plate, int modelYear, double tonnage) {
           this.plate = plate; this.modelYear = modelYear; this.tonnage = tonnage;
    }
    public double calculateAmortizedTax( double baseTax, int currentYear ) {
           double ratioTonnage, ratioAge;
           if( tonnage < 1.0 )
               ratioTonnage = 1.0;
           else if( tonnage < 5.0 )
               ratioTonnage = 1.2;
           else if( tonnage < 10.0 )
               ratioTonnage = 1.4;
           else
               ratioTonnage = 1.6;
           ratioAge = (currentYear - modelYear) * 0.05;
           if( ratioAge > 2.0 )
               ratioAge = 2.0;
           return baseTax * ratioTonnage * ratioAge;
    }
    public String getPlate() { return plate; }
    public int getModelYear() { return modelYear; }
    public double getEngineVolume() { return tonnage; }
}
```

DESIGNING AND CODING INTERFACES

- The state of a car, bus or a motorcycle instance being a personal or commercial vehicle will be saved in a Container object:



VehicleRegistrationSystem

- commercialVehicles[ ] : CommercialVehicle
- personalVehicles[ ] : PersonalVehicle

+ registerCommercialVehicle(CommercialVehicle) : boolean
+ registerPersonalVehicle(PersonalVehicle) : boolean
+ searchCommercialVehicle(plate : String) : CommercialVehicle
+ searchPersonalVehicle(plate : String) : PersonalVehicle
+ unregisterCommercialVehicle(plate : String) : boolean
+ unregisterPersonalVehicle(plate : String) : boolean

- How must the logic be? How should we implement that?

SPECIAL TOPICS in OOP

DESIGNING AND CODING INTERFACES

- If the tax calculation for different vehicles were similar (i.e. parametrized), using two abstract base classes instead of interfaces would be a better choice.
- Likewise, if the tax calculation for commercial and personal vehicles were similar, using only one abstract base class and choosing appropriate method parameters would be a better choice.
- Those cases are left as exercises to the students for experimenting with.

SPECIAL TOPICS in OOP

PRIMITIVE ENUMERATIONS (ENUMs)

- 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 …

SPECIAL TOPICS in OOP

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:

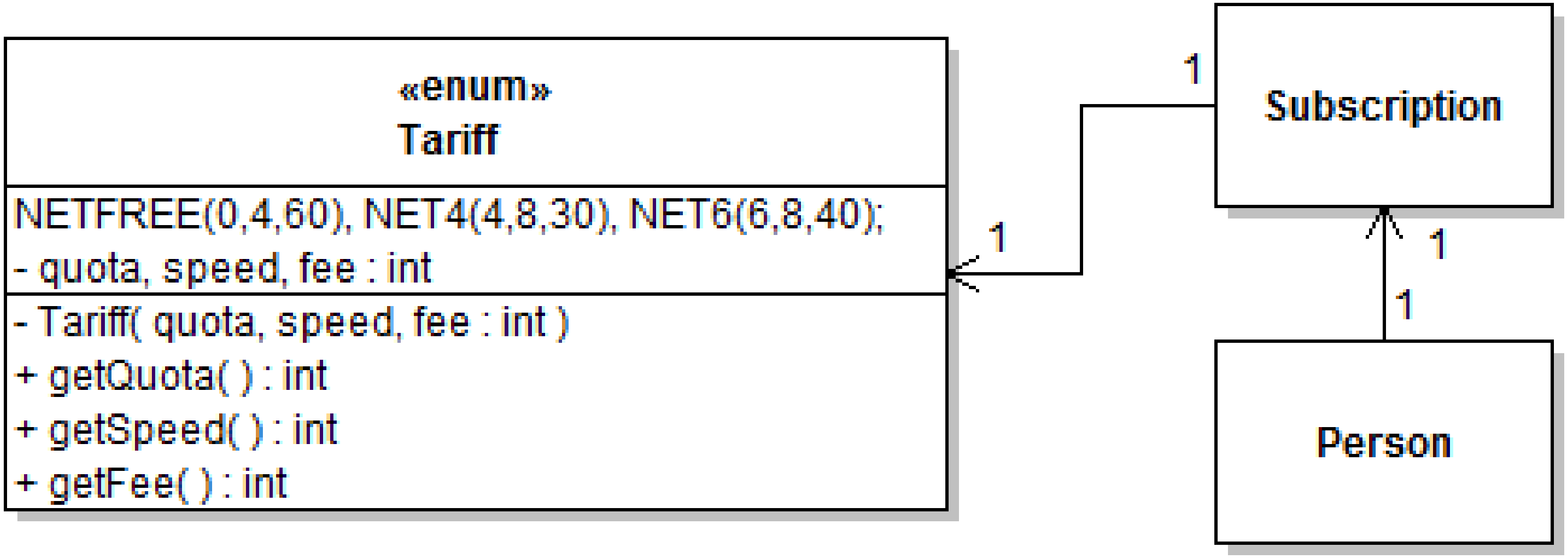
This must be the first line of code!

```
package nyp08;
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; }
}
```

SPECIAL TOPICS in OOP

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

SPECIAL TOPICS in OOP

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);
   }

}
```

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

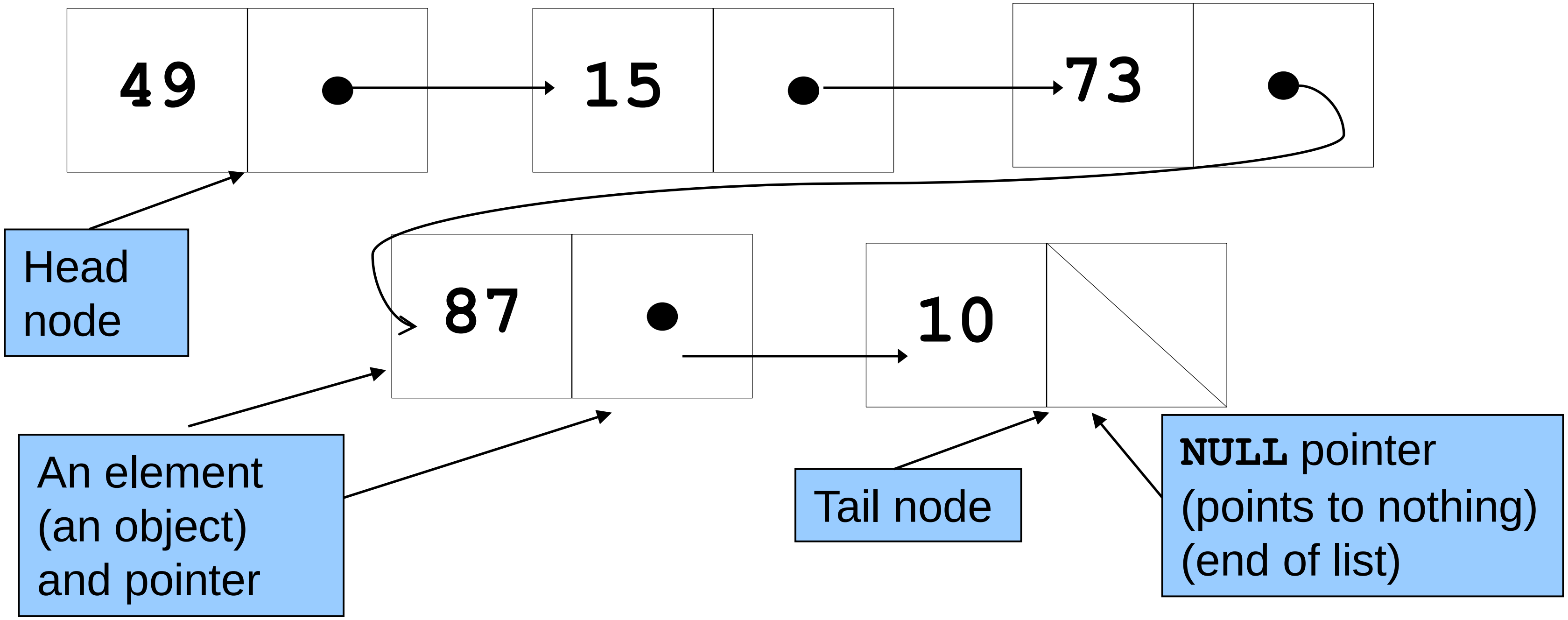
- Introduction to Data Structures:
  - A data structure is a scheme for organizing data in the memory of a computer
  - In a general sense, any data representation is a data structure.
    - Example: An integer.
  - More typically, a **data structure is** meant to be **an organization for a collection 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

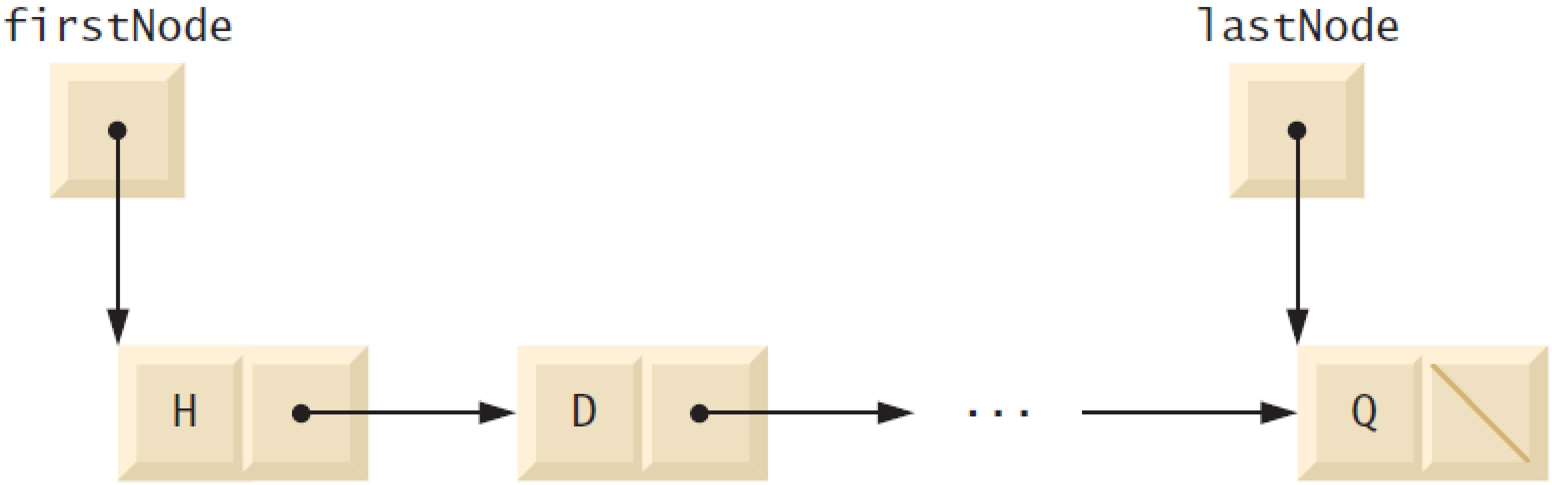
- Introduction to linked lists:
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links (transparent to the programming user in Java)
  - The first (head) and last (tail) nodes of the list are accessed via an object reference
  - Traversing between the nodes 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).

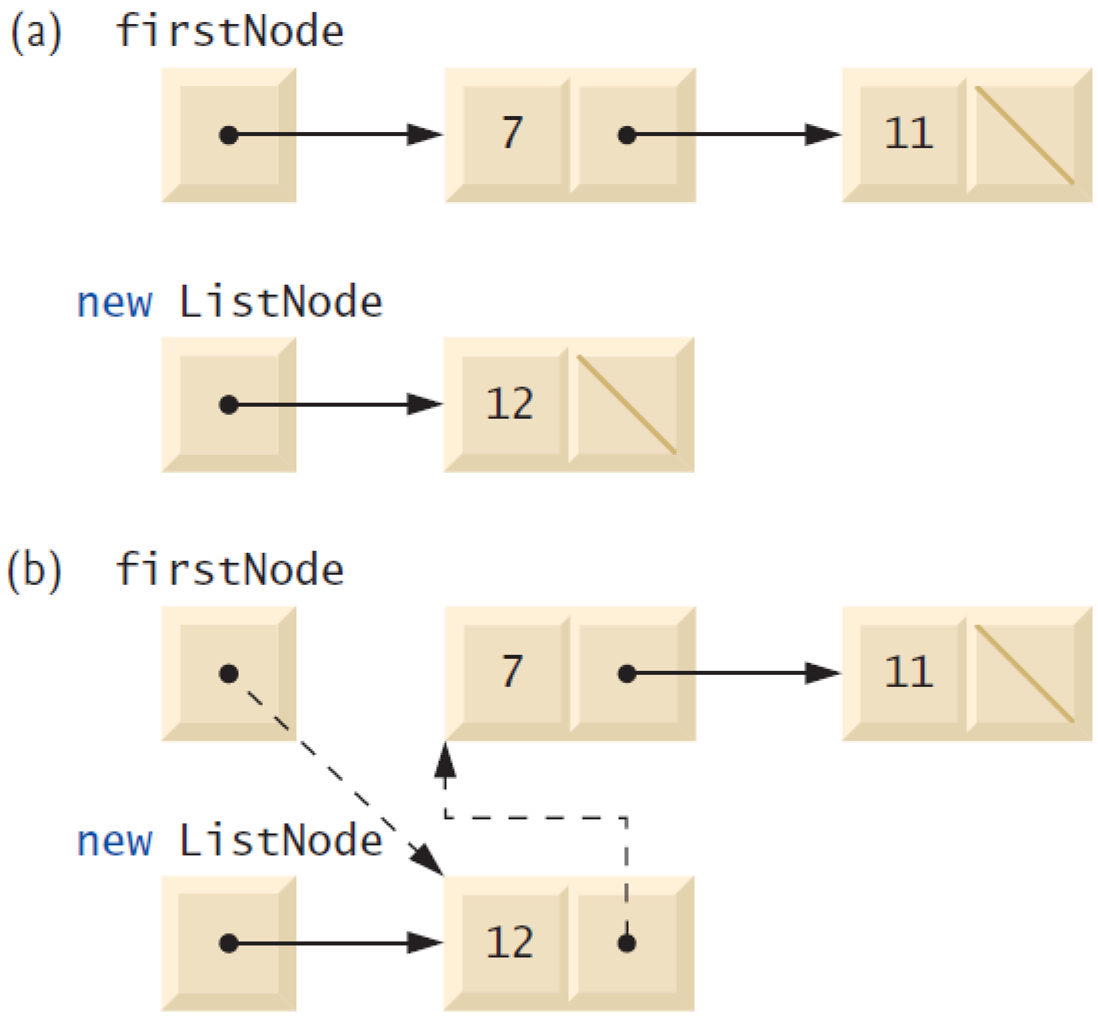
INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- An example linked list holding Integer objects:

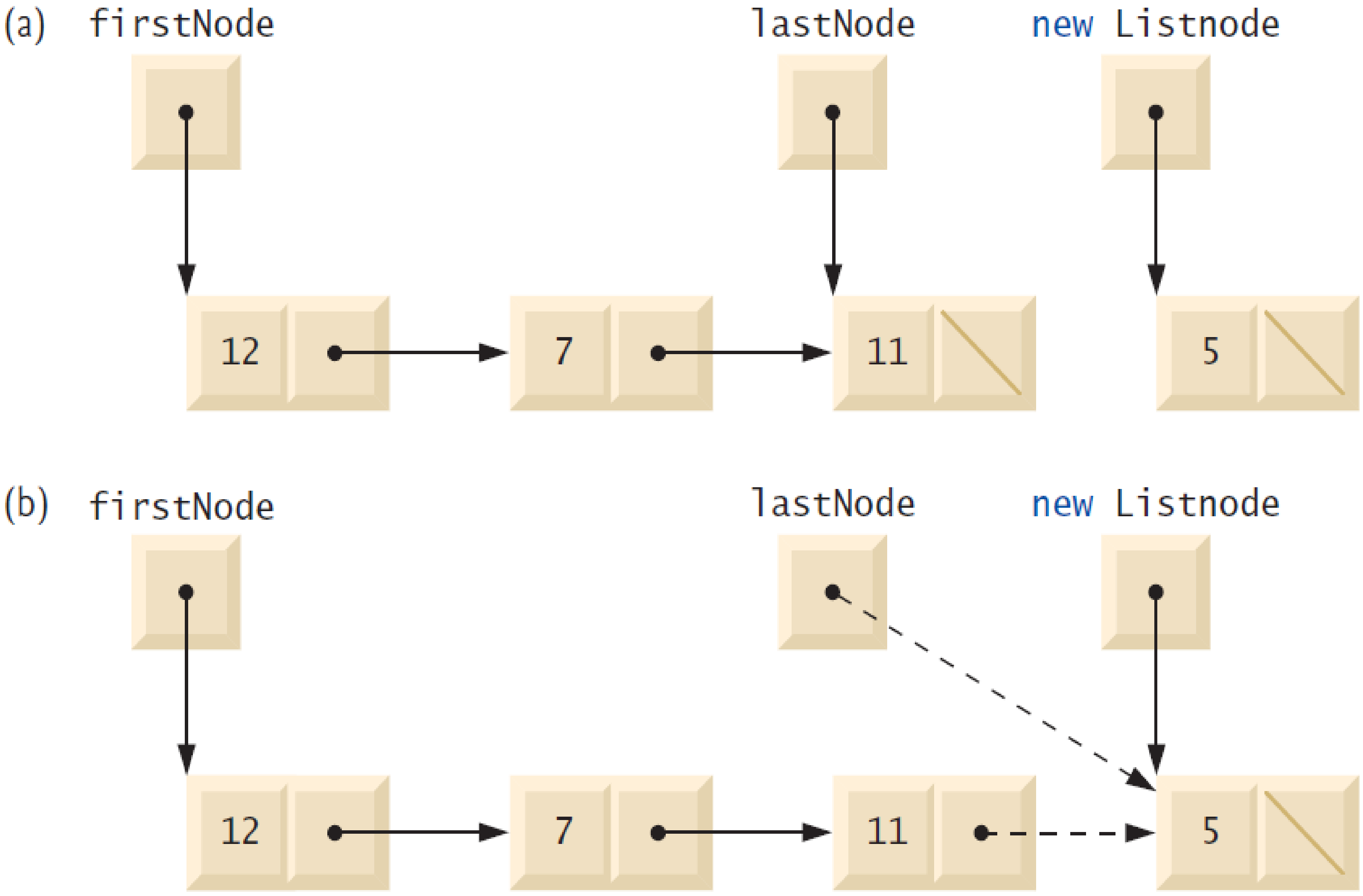

49
15
73
87
10
Head node
An element (an object) and pointer
Tail node
NULL pointer (points to nothing) (end of list)

- An example linked list holding String objects:



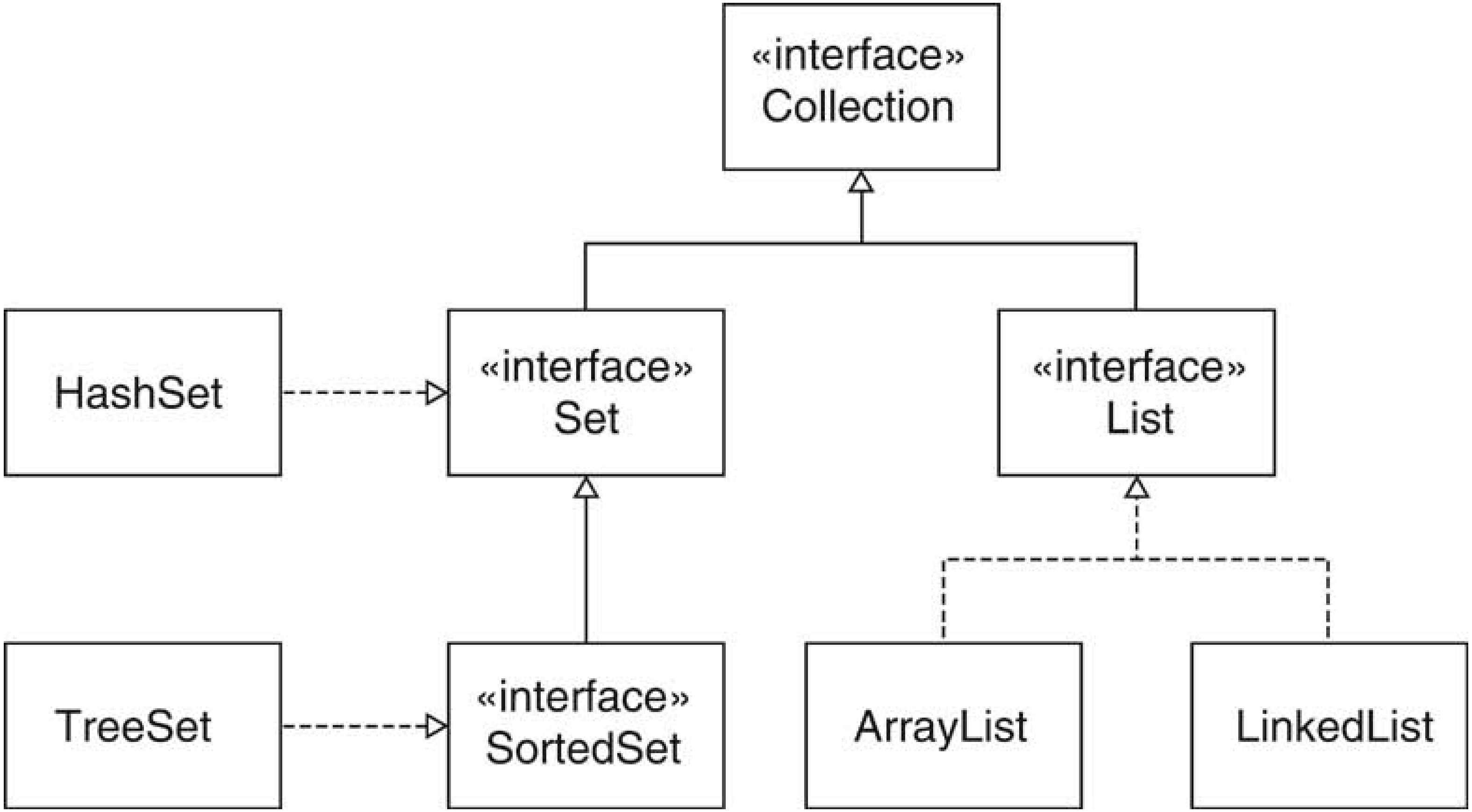


insertAtFront



insertAtBack

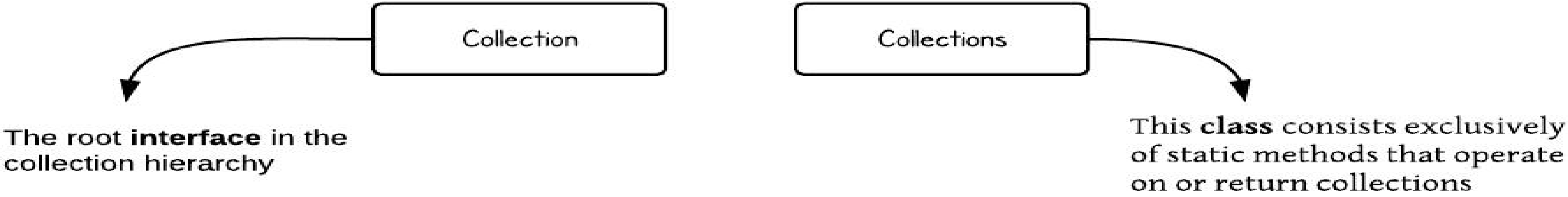
INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA



INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

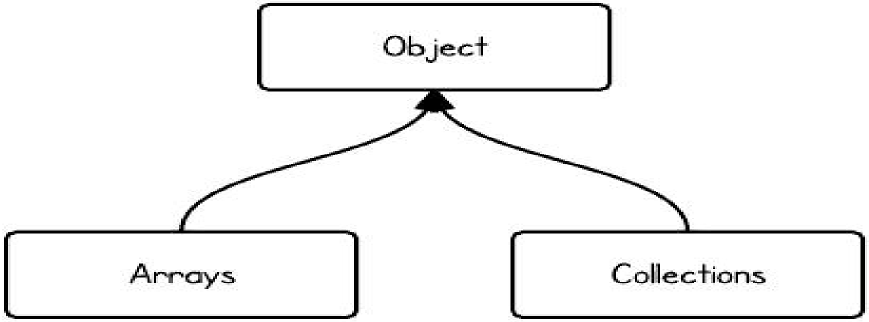
- Advantages of linked lists over arrays:
  - Enlarging a list costs nothing!
  - Insertion and removal of elements to any position is faster.
  - Sorting algorithms work faster on linked lists.
- Advantage of arrays over linked lists:
  - Lists are traversed sequentially where any i^{th} member of an 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 (like prayer beads)

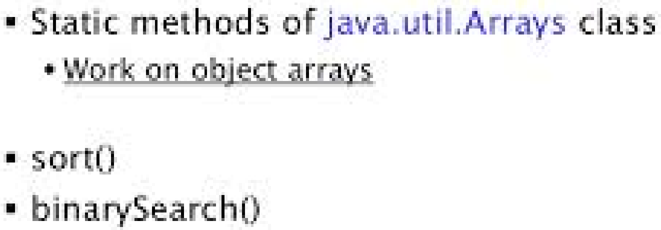
INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA



The root **interface** in the collection hierarchy

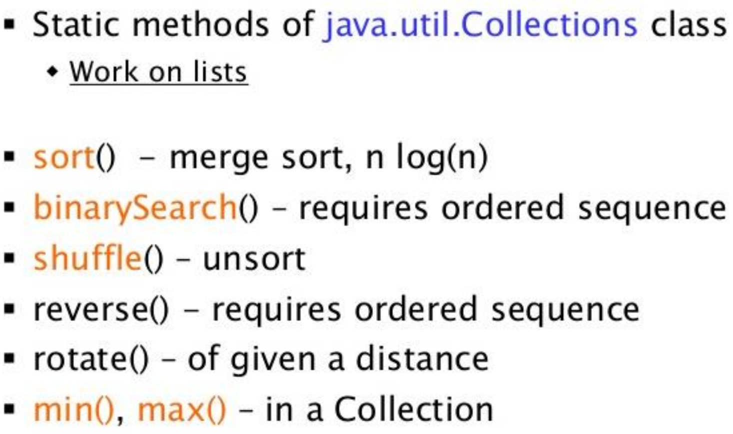
This **class** consists exclusively of static methods that operate on or return collections





- Static methods of java.util.Arrays class
  - Work on object arrays

- sort()
- binarySearch()
- **toArray(List list)**
- **asList(Array array)**

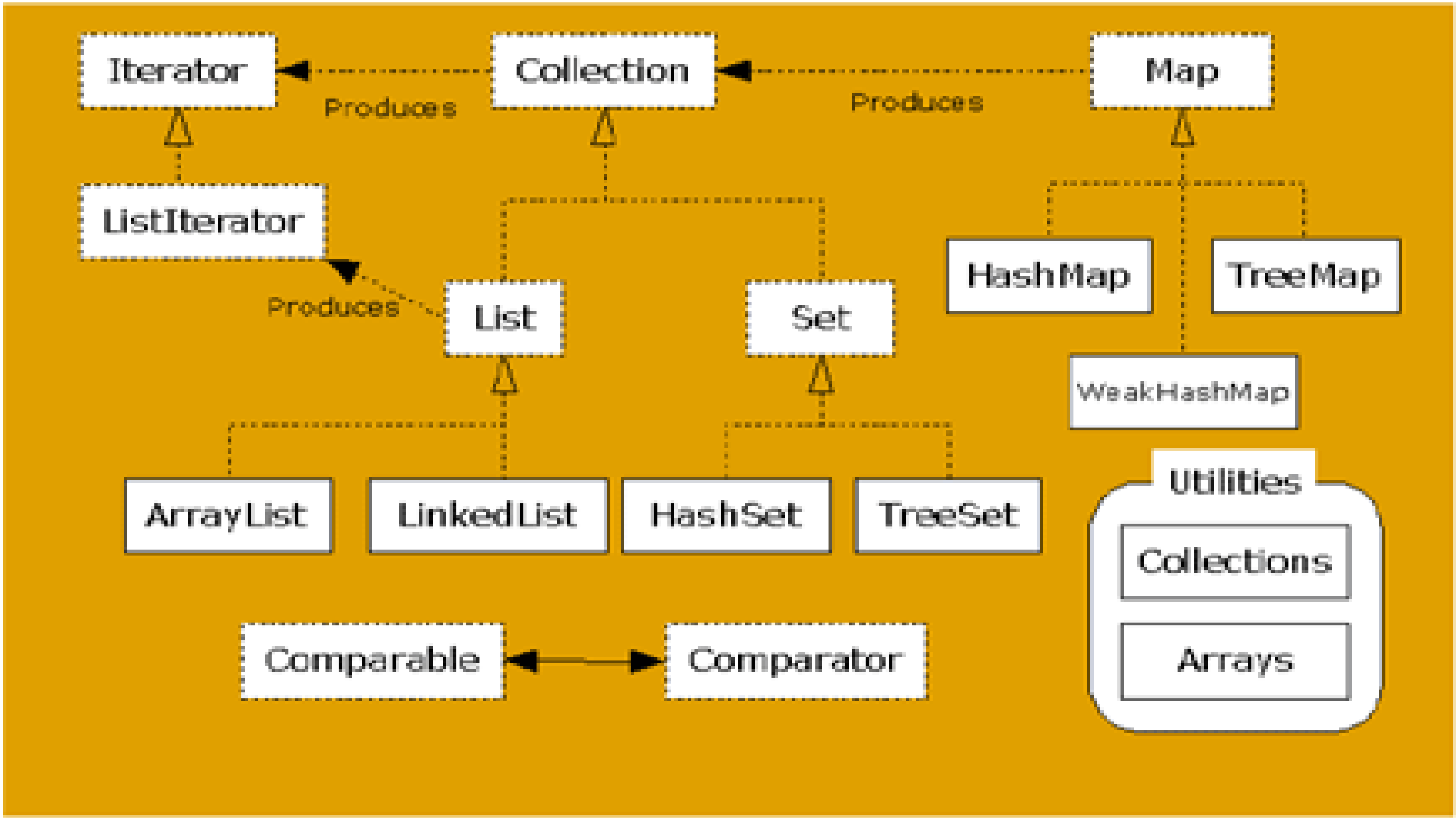


- Static methods of java.util.Collections class
  - Work on lists

- sort() – merge sort, n log(n)
- binarySearch() – requires ordered sequence
- shuffle() – unsort
- reverse() – requires ordered sequence
- rotate() – of given a distance
- min(), max() – in a Collection

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

Collection Interfaces and Classes



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

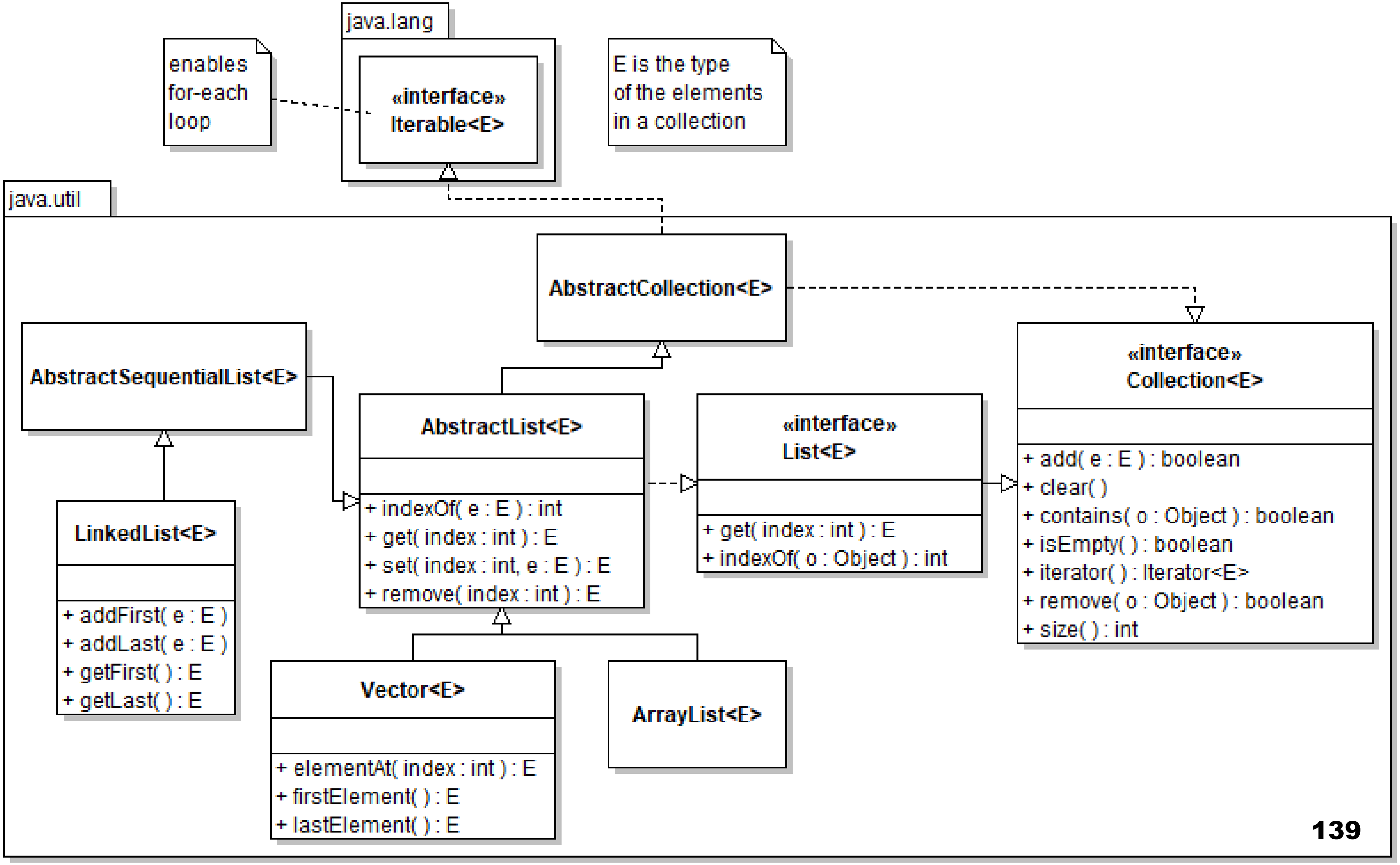
```
ArrayList<Person> liste = new ArrayList<Person>();
```

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).

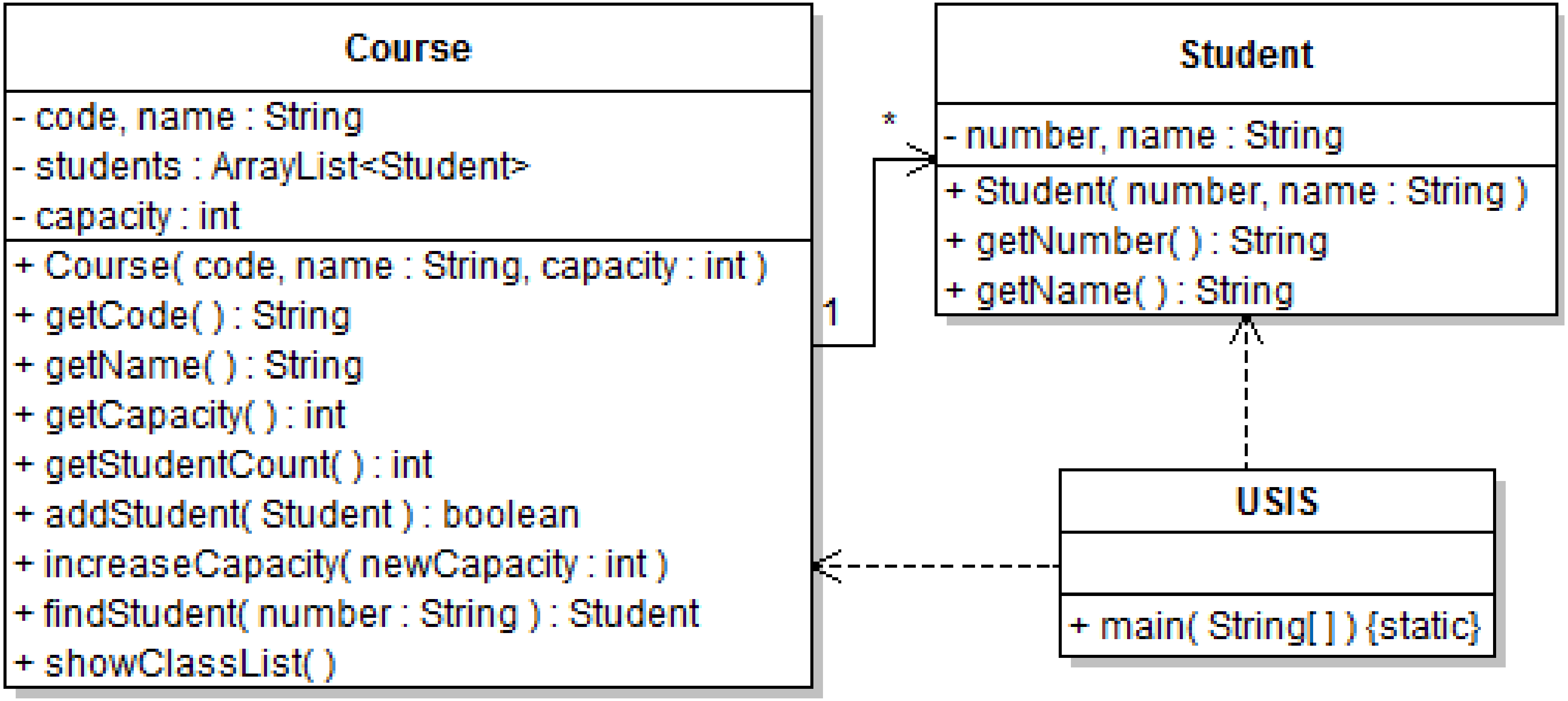
INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

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



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( )
1
*
Student
- number, name : String
+ Student(number, name : String)
+ getNumber( ) : String
+ getName( ) : String
USIS
+ main(String[ ]) {static}

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

```
package nyp09a;
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 nyp09x).
  - In this example, the member "capacity" exists only for business logic, not for array operations.

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

```
package nyp09x;
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());
    }
}
```

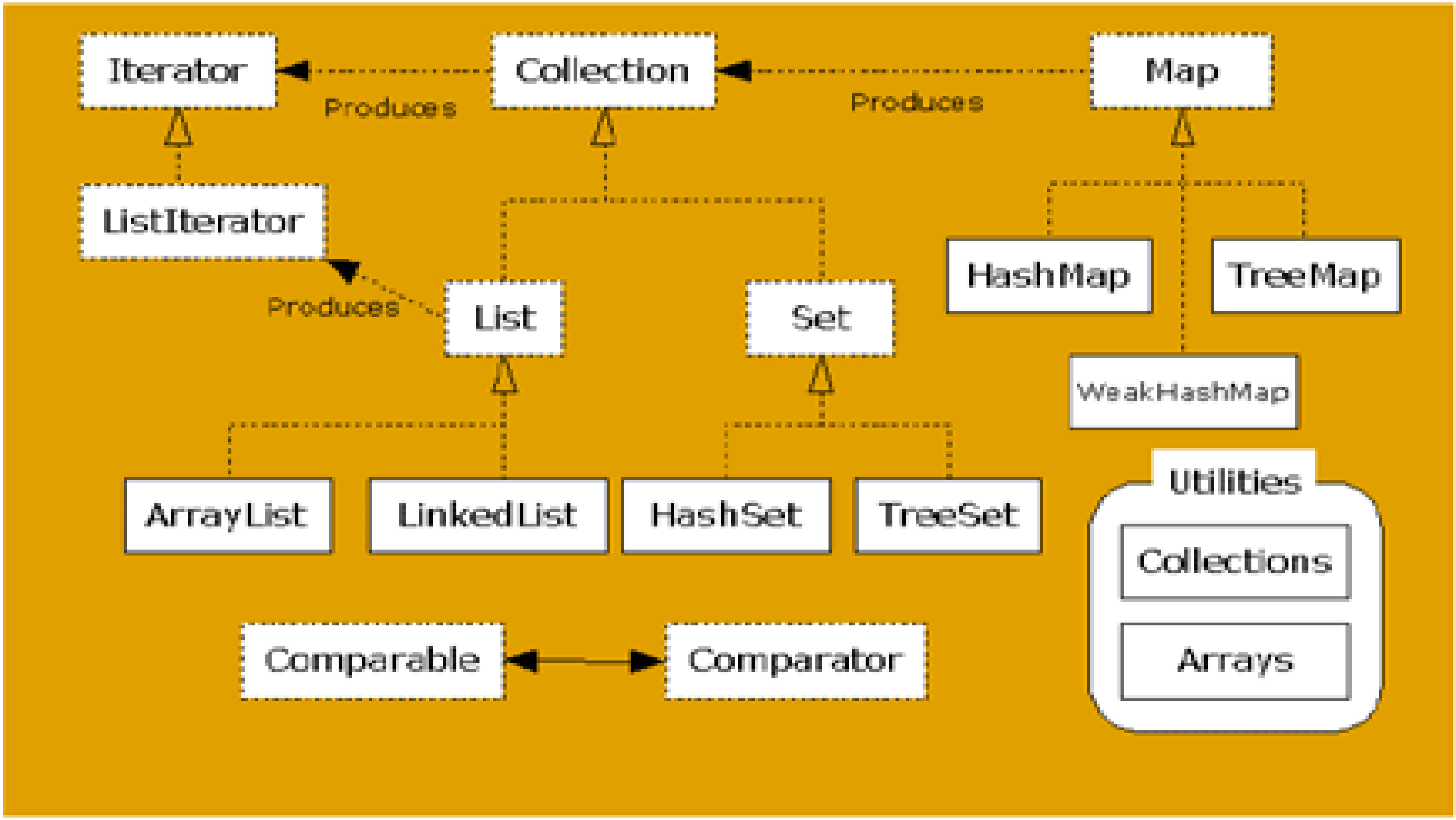
INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

- 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 yasar = new Student("09011034","Yaşar Nuri Öztürk");
         if( !oop.addStudent(yasar) )
         System.out.println("Problem #1");
         boolean result;
         result = oop.addStudent(yasar);
         if( result == true )
              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");
         if( oop.findStudent(fatih.getNumber()) == null )
              System.out.println("Problem #5");
         System.out.println("End of test\n");
         oop.showClassList();
     }
}
```

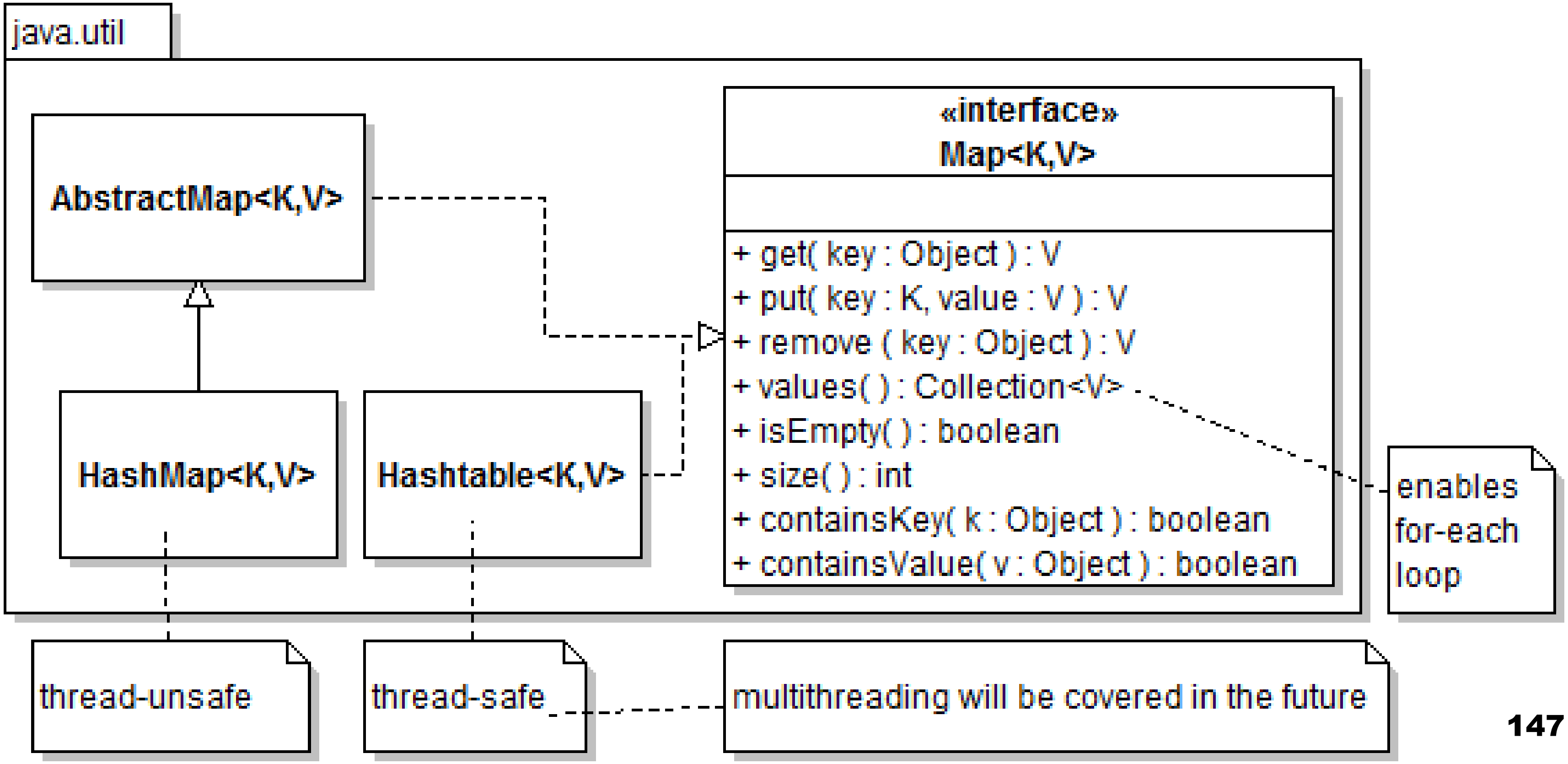
INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

Collection Interfaces and Classes



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
AbstractMap<K,V>
«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
HashMap<K,V>
Hashtable<K,V>
enables for-each loop
thread-unsafe
thread-safe
multithreading will be covered in the future

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 nyp09b;
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");
    }
}
```

INTRO. GENERIC CLASSES and DATA STRUCTURES in JAVA

SUMMARY OF FUNDAMENTAL DATA STRUCTURE IMPLEMENTATIONS:

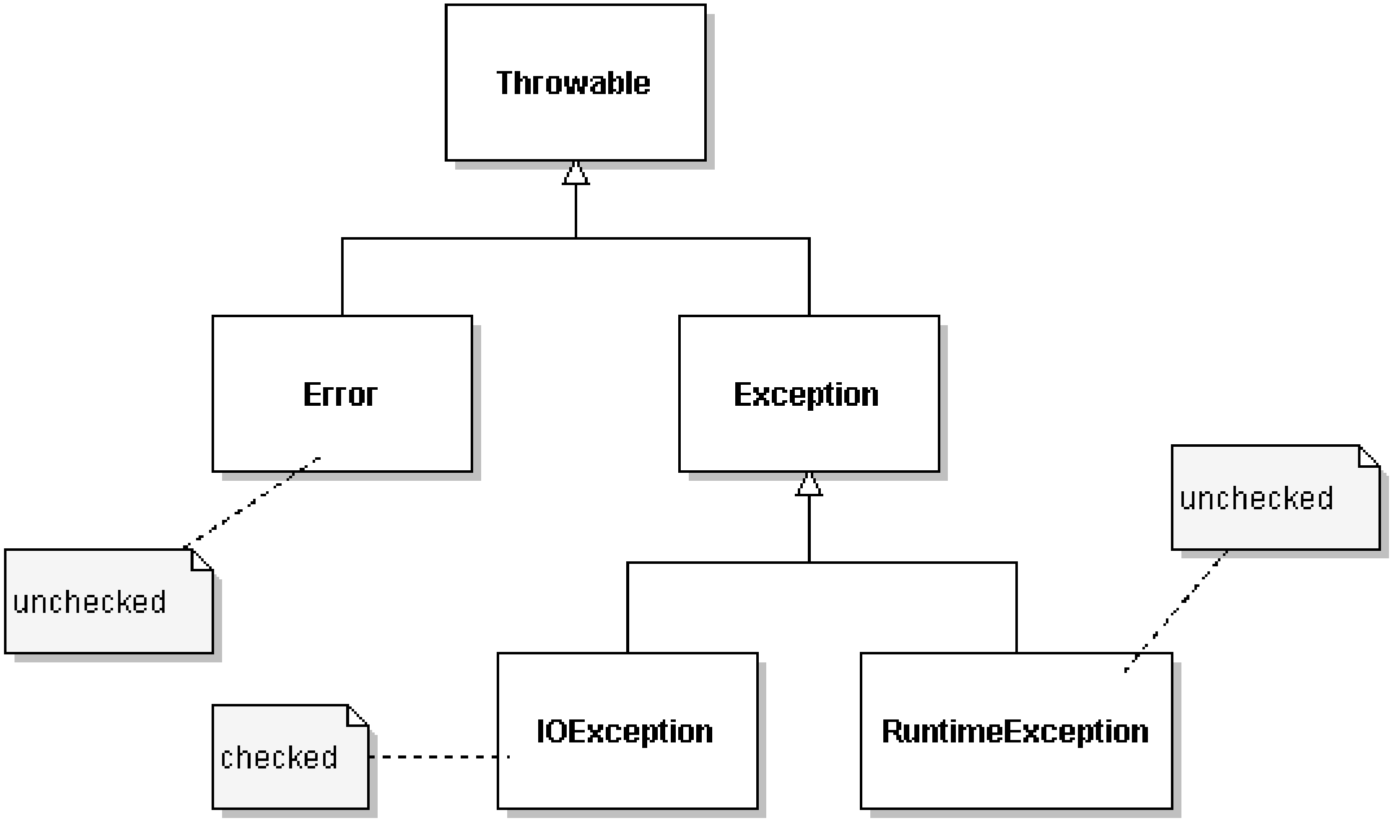
- java.util.LinkedList<E> implements List<E>
  - Faster insertions and deletions
  - Slower random access
  - Doubly-linked (Can be traversed backwards by obtaining a ListIterator instance [not to be covered?]).
- java.util.ArrayList<E> implements List<E>
  - Slower insertions and deletions
  - Faster 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

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 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); //Multithreaded, allows finally to be run
        }
        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 );
        /* Other things to do (optional) */
    } //necessary for informing the user and/or programmer
}
```

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 nyp10;
import java.io.IOException;
/* @home: Check the needed additions
 * if we had extended this exception
 * from java.lang.RuntimeException
 */
@SuppressWarnings("serial")
public class ImpossibleInfo extends IOException {
       public ImpossibleInfo( String errorMessage ) {
              super(errorMessage);
       }
    }
}
```

EXCEPTION HANDLING

```
package nyp10;
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 nyp10;
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();
        }
    }
}
```

Exercise: Check for too short names

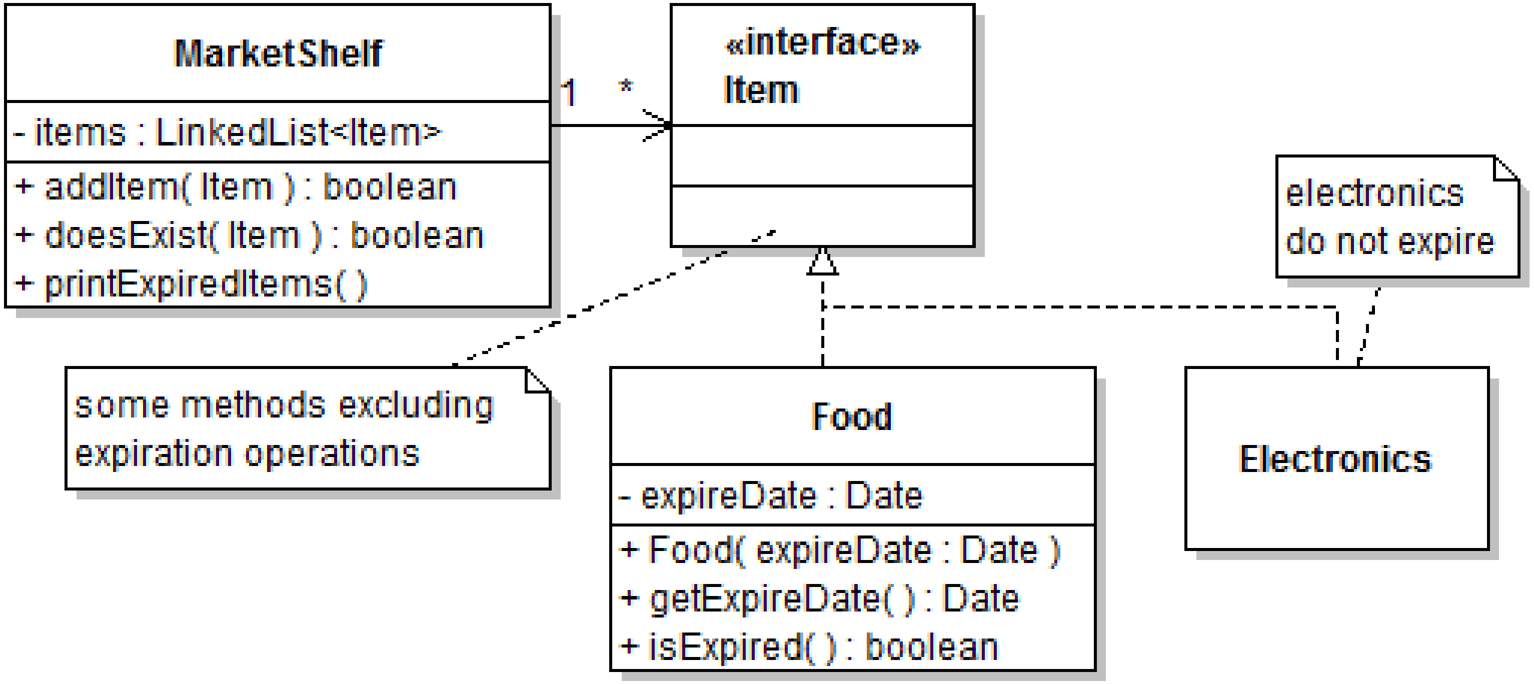
- You can wrap the statement that can cause an exception and the remaining statements with the try block or you can put all statements into the try block.

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:



MarketShelf

- items : LinkedList<Item>

+ addItem(Item) : boolean

+ doesExist(Item) : boolean

+ printExpiredItems( )

1 *

«interface»
Item

electronics
do not expire

some methods excluding
expiration operations

Food

- expireDate : Date

+ Food(expireDate : Date)

+ getExpireDate( ) : Date

+ isExpired( ) : boolean

Electronics

TYPECASTING

- Coding class MarketShelf:

```
package nyp11;
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!");
}
```

- Checking for type compliance and typecasting

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, 13);
        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

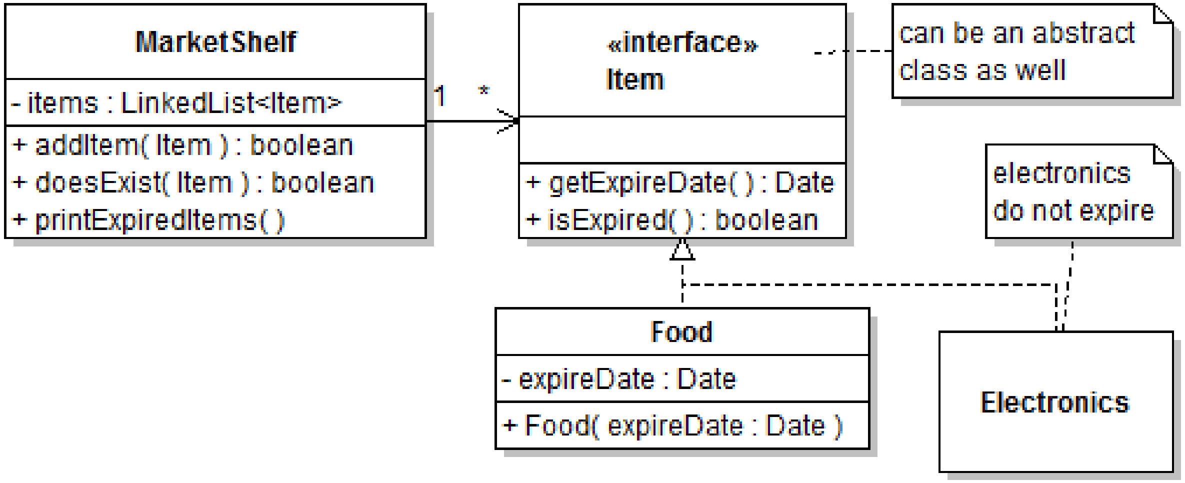
- Coding class Food:

```
package nyp11;
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.
- About the **`transient`** keyword
  - Mark a member fields of a class as transient if you do not want to serialize it.
  - You will need to do so if the class having that member has to be serialized but you cannot mark that member's class as Serializable.

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Example record: the class Contact

```
package nyp12a;
public class Contact implements java.io.Serializable {
   private static final long serialVersionUID = 1L;
   private String name, phone, eMail;
   public Contact( String name ) { this.name = name; }
   public String getName( ) { return name; }
   public String getPhone( ) { return phone; }
   public void setPhone( String telefon ) {
      this.phone = telefon;      }
   public String getEMail( ) { return eMail; }
   public void setEMail( String mail ) { eMail = mail; }
   public String toString( ) {
      return name + " - " + phone + " - " + eMail;
   }
}
```

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):

```
package nyp12a;
import java.util.*;
import java.io.*;
public class CreateContacts {
   public static void main(String[] args) {
      Scanner input = new Scanner( System.in );
      System.out.println("This program saves information of your " +
                          " contacts to a file on your drive.");
      System.out.print("How many contacts will you enter? ");
      Integer contactCount = input.nextInt( );
      Contact[] contacts = new Contact[contactCount];
      input.nextLine( );
      for( int i = 0; i < contactCount; i++ ) {
         System.out.print("What is the name of the contact #"+(i+1)+"? ");
         contacts[i] = new Contact( input.nextLine() );
         System.out.print("What is the phone number of this contact? ");
         contacts[i].setPhone( input.nextLine() );
         System.out.print("What is the e-mail address of this contact? ");
         contacts[i].setEMail( input.nextLine() );
      }
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Serialization example (cont'd):

```
      try {
          String fileName = "contacts.dat";
          ObjectOutputStream writer = new ObjectOutputStream(
               new FileOutputStream( fileName, true )  );
          writer.writeObject( contactCount );
          for( Contact aContact : contacts )
               writer.writeObject( aContact );
          writer.close();
          System.out.println("The information you have entered has "
               + "been successfully saved in file " + fileName);
      }
      catch( IOException e ) {
          System.out.println("An exception has occured during "
               + "writing to file.");
          e.printStackTrace();
      }
      input.close();
   }
}//HW: Do not use an array
```

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.
        - The warning “Type safety: Unchecked cast” can be suppressed by @SuppressWarnings("unchecked")
    - 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):

```
package nyp12a;
import java.io.*;
public class ShowContacts {
  public static void main( String[] args ) {
    String fileName = "contacts.dat";
    try {
        ObjectInputStream reader = new ObjectInputStream(
                       new FileInputStream( fileName ) );
        Integer contactCount = (Integer) reader.readObject();
        for( int i=0; i<contactCount; i++ ) {
               Contact aContact = (Contact)reader.readObject();
               System.out.println(aContact);
        }
        reader.close();
    }
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Deserialization example (cont’d):

```
    catch( IOException e ) {
        System.out.println("A file reading exception has occured.");
        e.printStackTrace();
    }
    catch( ClassNotFoundException e ) {
        System.out.println("A class cast exception has occured.");
        e.printStackTrace();
    }
  }
}//Alternative: You could populate an array, too.
```

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 reader = new ObjectInputStream(
                    new FileInputStream( fileName ) );
    Contact aContact = (Contact) reader.readObject();
    while(reader.hasNext()) {
            System.out.println(aContact);
            aContact = (Contact) reader.readObject();
    }
    reader.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 (PS: Contact class stays the same):

```
package nyp12b;
import java.util.*;
import java.io.*;
public class CreateContacts {
   public static void main(String[] args) {
      LinkedList<Contact> contacts = new LinkedList<Contact>();
      Scanner input = new Scanner( System.in );
      System.out.println("This program saves information of your " +
                         " contacts to a file on your drive.");
      System.out.print("How many contacts will you enter? ");
      int contactCount = input.nextInt( );
      input.nextLine( );
      for( int i = 0; i < contactCount; i++ ) {
          System.out.print("What is the name of the contact #"+(i+1)+"? ");
          Contact aContact = new Contact( input.nextLine() );
          System.out.print("What is the phone number of this contact? ");
          aContact.setPhone( input.nextLine() );
          System.out.print("What is the e-mail address of this contact? ");
          aContact.setEMail( input.nextLine() );
          contacts.add(aContact);
      }
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Serializing data structures (cont'd):

```
      try {
          String fileName = "contacts.dat";
          ObjectOutputStream yazici = new ObjectOutputStream(
                 new FileOutputStream( fileName, true )   );
          yazici.writeObject( contacts );
          yazici.close();
          System.out.println("The information you have entered has "
                 + "been successfully saved in file " + fileName);
      }
      catch( IOException e ) {
          System.out.println("An exception has occured during " +
                 "writing to file.");
          e.printStackTrace();
      }
      input.close();
   }
}
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Deserializing data structures:

```
package nyp12b;
import java.io.*;
import java.util.*;
public class ShowContacts {
@SuppressWarnings("unchecked")
    public static void main( String[] args ) {
         String fileName = "contacts.dat";
         try {
            ObjectInputStream reader = new ObjectInputStream(
                 new FileInputStream( fileName ) );
            LinkedList<Contact> contacts =
                 (LinkedList<Contact>)reader.readObject();
            for( Contact aContact : contacts ) {
                 System.out.println(aContact);
            }
            reader.close();
         }
```

WORKING WITH FILES

I/O OPERATIONS USING STREAMS

- Deserializing data structures (cont'd.):

```
    catch( IOException e ) {
        System.out.println("An exception has occured during file reading.");
        e.printStackTrace();
    }
    catch( ClassNotFoundException e ) {
        System.out.println("An exception has occured while processing.");
        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.

INNER CLASSES

- You can code a class **within** 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

```java
package nyp13a;
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 nyp13b;

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 nyp13b;

//this import is absolutely necessary
import nyp13b.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 nyp13c;
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 nyp13c;
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 );
    }
}
```

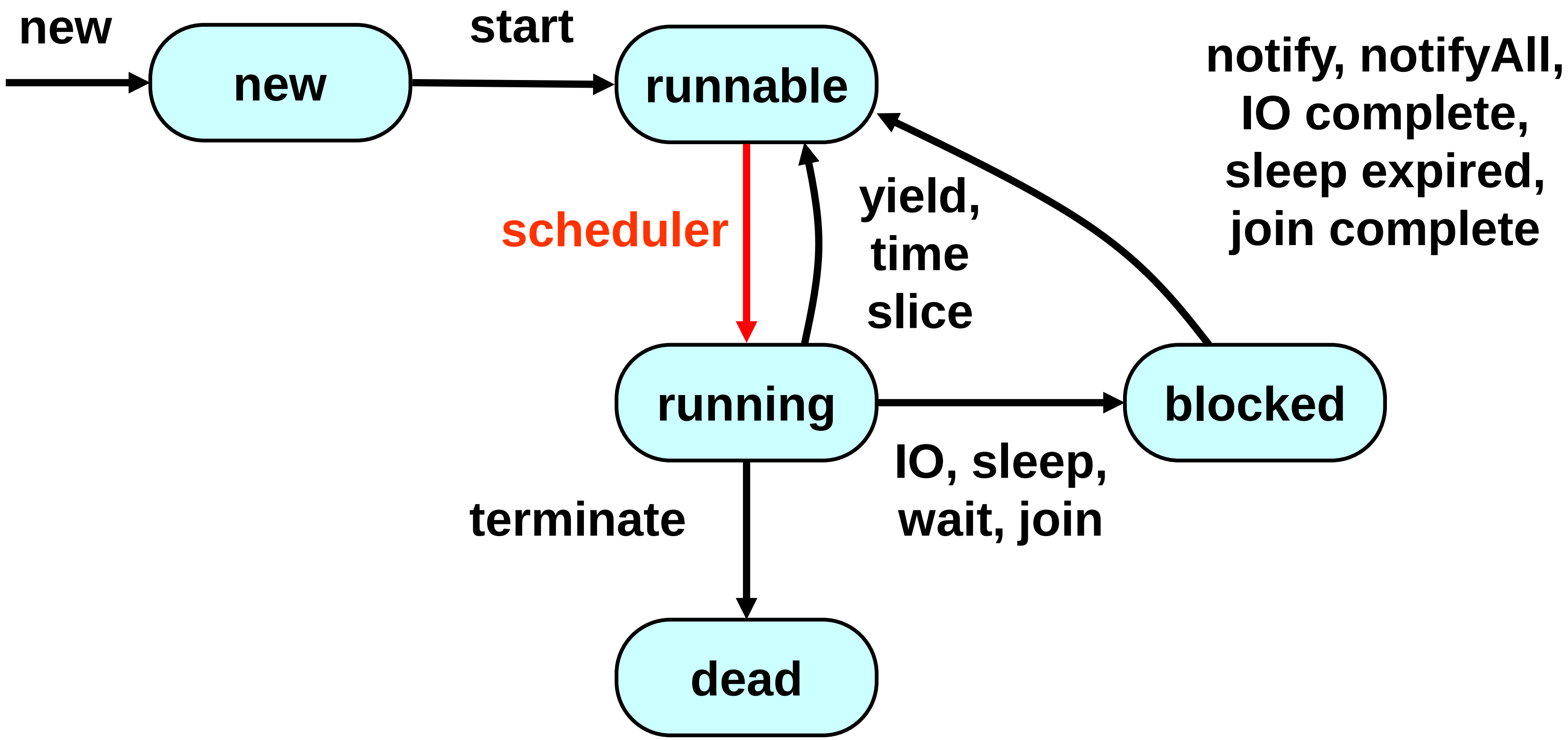
- You can obtain a reference to the outer class instance from the inner class instance by:
  - OuterClassName.this
  - In this case, you can use Person.this from the Employee inner class.
- However, this type of access is rarely needed as the inner class instance can use even the private methods of the outer class instance.

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 nyp14a;
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

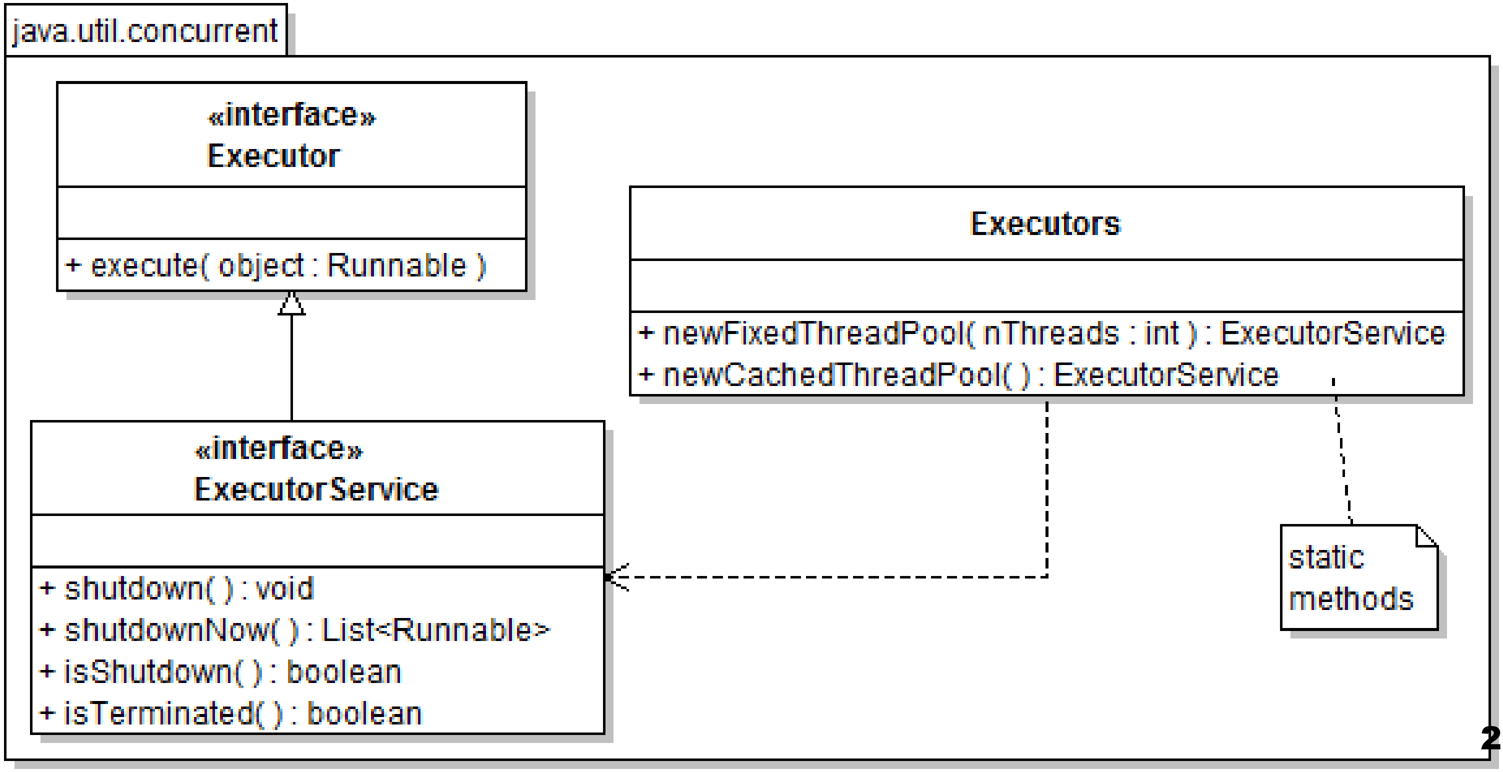
2. Create an object
of your class

```
package nyp14a;

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( );
    }
}
```

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)

Executors

+ newFixedThreadPool(nThreads : int) : ExecutorService
+ newCachedThreadPool( ) : ExecutorService

«interface»
ExecutorService

+ shutdown( ) : void
+ shutdownNow( ) : List<Runnable>
+ isShutdown( ) : boolean
+ isTerminated( ) : boolean

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 nyp14a;
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

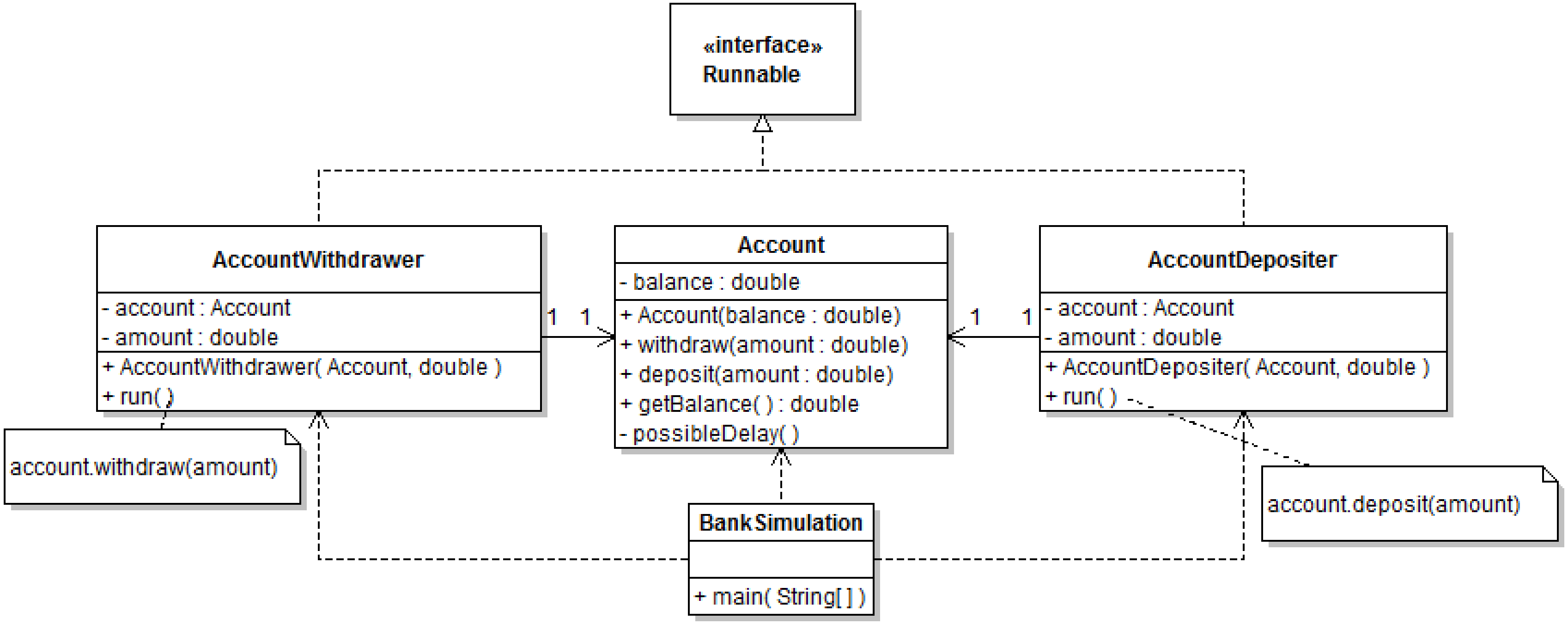
- Exceptions and multithreading:
  - Throwing an unchecked exception from the run() method is easy.
  - You cannot change the method signature of the run method to declare that an unchecked exception can be thrown.
  - To throw a checked exception from the run() method, you need to:
    1. Code the multithreaded task that can throw the exception in a normal member method
    2. Declare that method as **`throws SomeCheckedException`**
    3. Call that method from run() and use try/catch properly.

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 nyp14b;
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

```java
package nyp14b;
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 nyp14b;
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

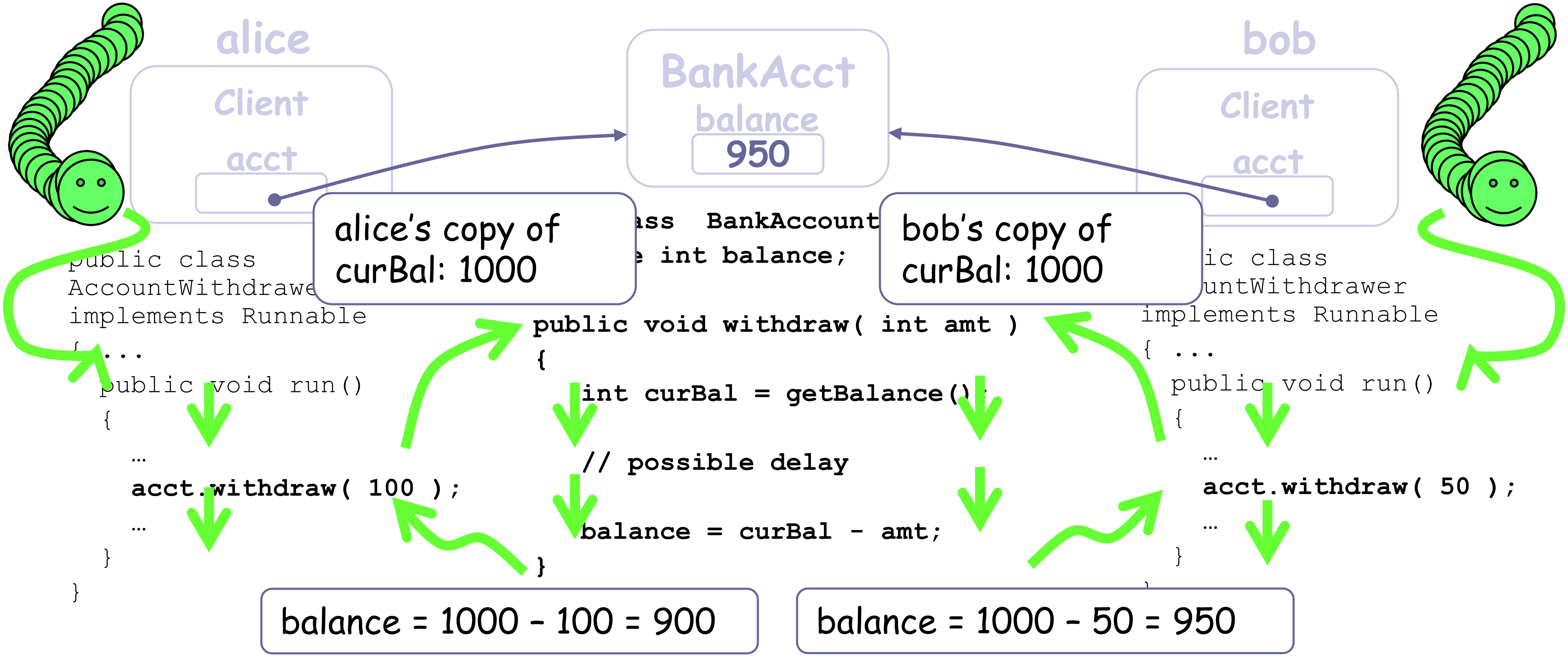
```
package nyp14b;
import java.util.concurrent.*;
public class BankSimulation {
    public static void main(String[] args) {
        Account anAccount = new Account(0);
        System.out.println("Before: "+anAccount.getBalance());
        ExecutorService executor = Executors.newCachedThreadPool( );
        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 );
    …
  }
}
```

```
ass  BankAccount
e int balance;

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

  // possible delay

  balance = curBal - amt;
}
```

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

alice's copy of curBal: 1000

bob's copy of curBal: 1000

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 nyp14c;
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.

This slide is intentionally left blank