OOP - JAVA

Object-Oriented Programming
Java

Margit ANTAL
Sapientia Hungarian University of Transylvania
2023

Goals

1. Java Language
2. Objects and classes
3. Static Members
4. Relationships between classes
5. Inheritance and Polymorphism
6. Interfaces and Abstract Classes
7. Exceptions
8. Nested Classes
9. Threads
10. GUI Programming
11. Collections and Generics
12. Serialization

Module 1
Java language

Java language

- History
- Java technology: JDK, JRE, JVM
- Properties
- *Hello world* application
- Garbage Collection

Short History

- 1991 - Green Project for consumer electronics market (Oak language → Java)
- 1994 – HotJava Web browser
- 1995 – Sun announces Java
- 1996 – JDK 1.0
- 1997 – JDK 1.1 *RMI, AWT, Servlets*
- 1998 – Java 1.2 *Reflection, Swing, Collections*
- 2004 – J2SE 1.5 (Java 5) *Generics, enums*
- 2014 – Java SE 8 *Lambdas - functional programming*

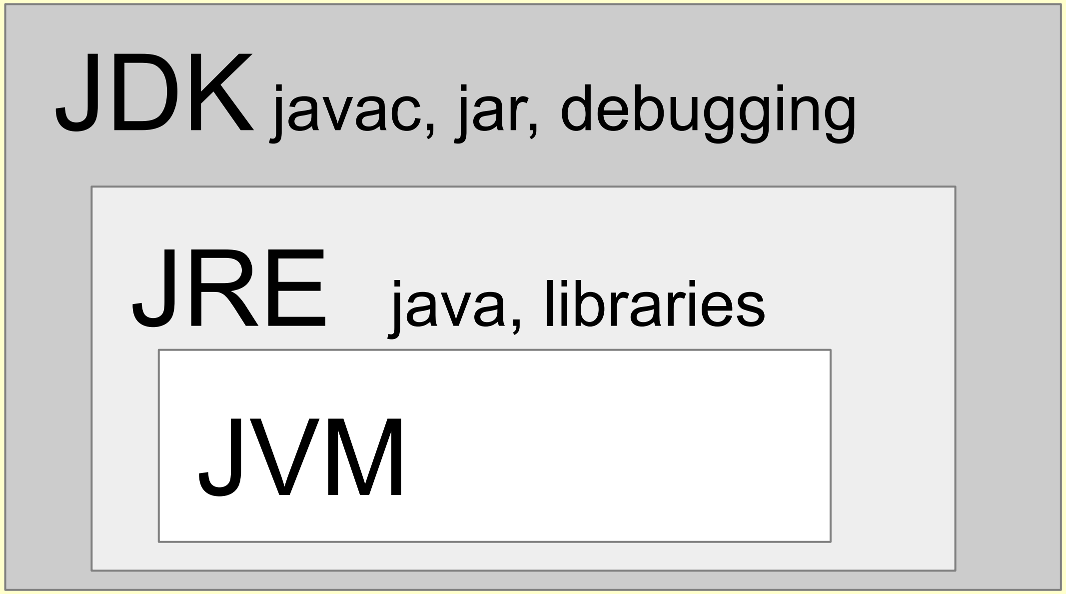
Short History

https://en.wikipedia.org/wiki/Java_version_history

- 2017 - Java SE 9
- 2018 - Java SE 10, Java SE 11
- 2019 - Java SE 12, Java SE 13
- 2020 - Java SE 14, Java SE 15
- 2021 - Java SE 16, Java SE 17
- 2022 - Java SE 18, Java SE 19
- 2023 - Java SE 20

Java technology

- JDK – Java Development Kit
- JRE – Java Runtime Environment
- JVM – Java Virtual Machine



Properties

- Object-oriented
- Interpreted
- Portable
- Secure and robust
- Scalable
- Multi-threaded
- Dynamic capabilities (reflection)
- Distributed

Hello World Application

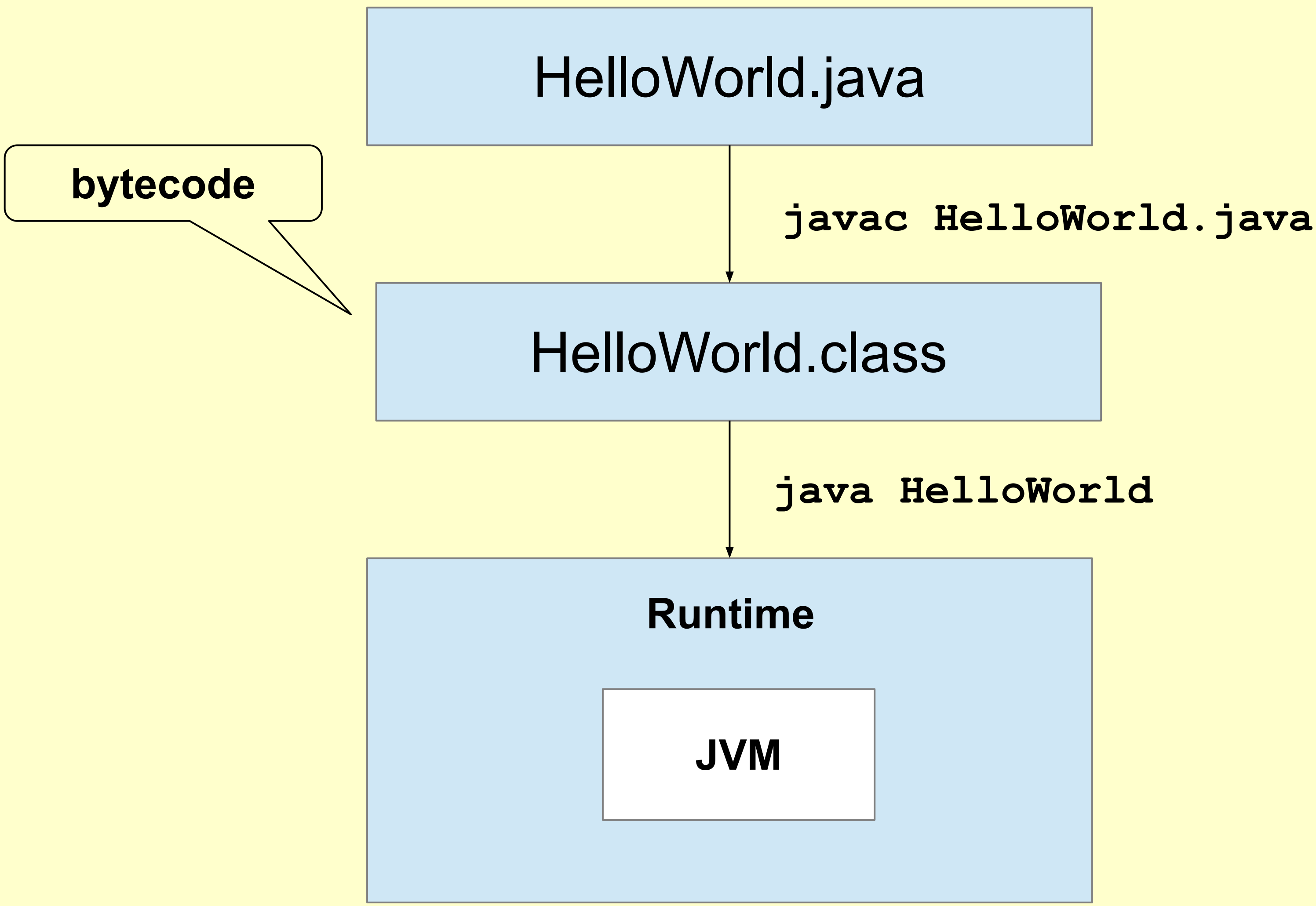
1. Write the source code: HelloWorld.java

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

2. Compile: `javac HelloWorld.java`

3. Run: `java HelloWorld`

Hello World Application



Garbage Collection

- Dynamically **allocated memory**
- **Deallocation**
  - Programmer's responsibility (C/C++)
  - System responsibility (Java):
    - Is done automatically (system-level thread)
    - Checks for and frees memory no longer needed

Remember

- JVM, JRE, JDK
- Compilers vs. interpreters
- Portability

Module 2
Object-Oriented Programming

Object-oriented programming

Classes and Objects

- Class
- Attributes and methods
- Object (instance)
- Information hiding
- Encapsulation
- Constructors
- Packages

Class

- Is a **user-defined type**
  - Describes the *data* **(attributes)**
  - Defines the *behavior* (**methods**)

members

- Instances of a class are **objects**

Declaring Classes

- **Syntax**

```
<modifier>* class <class_name>{
    <attribute_declaration>*
    <constructor_declaration>*
    <method_declaration>*
}
```

- **Example**

```
public class Counter {
   private int value;
   public void inc(){
      ++value;
   }
   public int getValue(){
      return value;
   }
}
```

Declaring Attributes

- **Syntax**

  `<modifier>* <type> <attribute_name>[= <initial_value>];`

- **Examples**

```
public class Foo {
    private int x;
    private float f = 0.0;
    private String name =”Anonymous”;
}
```

Declaring Methods

- **Syntax**

```
<modifier>* <return_type> <method_name>( <argument>* ){
   <statement>*
}
```

- **Examples**

```
public class Counter {
   public static final int MAX = 100;
   private int value;

   public void inc(){
      if( value < MAX ){
         ++value;
      }
   }
   public int getValue(){
      return value;
   }
}
```

Accessing Object Members

- **Syntax**

`<object>.<member>`

- **Examples**

```
public class Counter {
   public static final int MAX = 100;
   private int value = 0;

   public void inc(){
      if( value < MAX ){
         ++value;
      }
   }
   public int getValue(){
      return value;
   }
}
```

```
Counter c =  new   Counter();
c.inc();
int i = c.getValue();
```

Information Hiding

- **The problem:**

Client code has direct access to internal data

```
/* C language */
struct Date {
   int year, month, day;
};
```

```
/* C language */
 Date d;
 d.day = 32; //invalid day

 d.month = 2; d.day = 30;
 // invalid data

 d.day = d.day + 1;
 // no check
```

Information Hiding

- **The solution:**

Client code must use setters and getters to access internal data

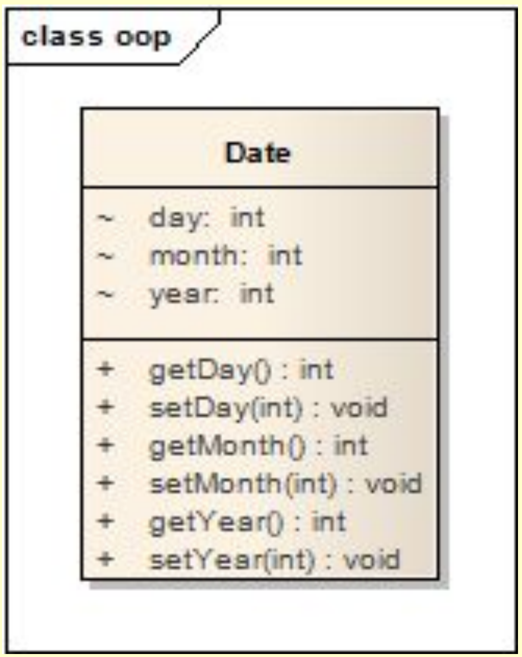
```
// Java language
public class Date {
   private int year, month, day;
   public void setDay(int d){..}
   public void setMonth(int m){..}
   public void setYear(int y){..}
   public int getDay(){...}
   public int getMonth(){...}
   public int getYear(){...}
}
```

```
Date d = new Date();
// no assignment
d.setDay(32);
// month is set
d.setMonth(2);
// no assignment
d.day = 30;
```

Verify days in month

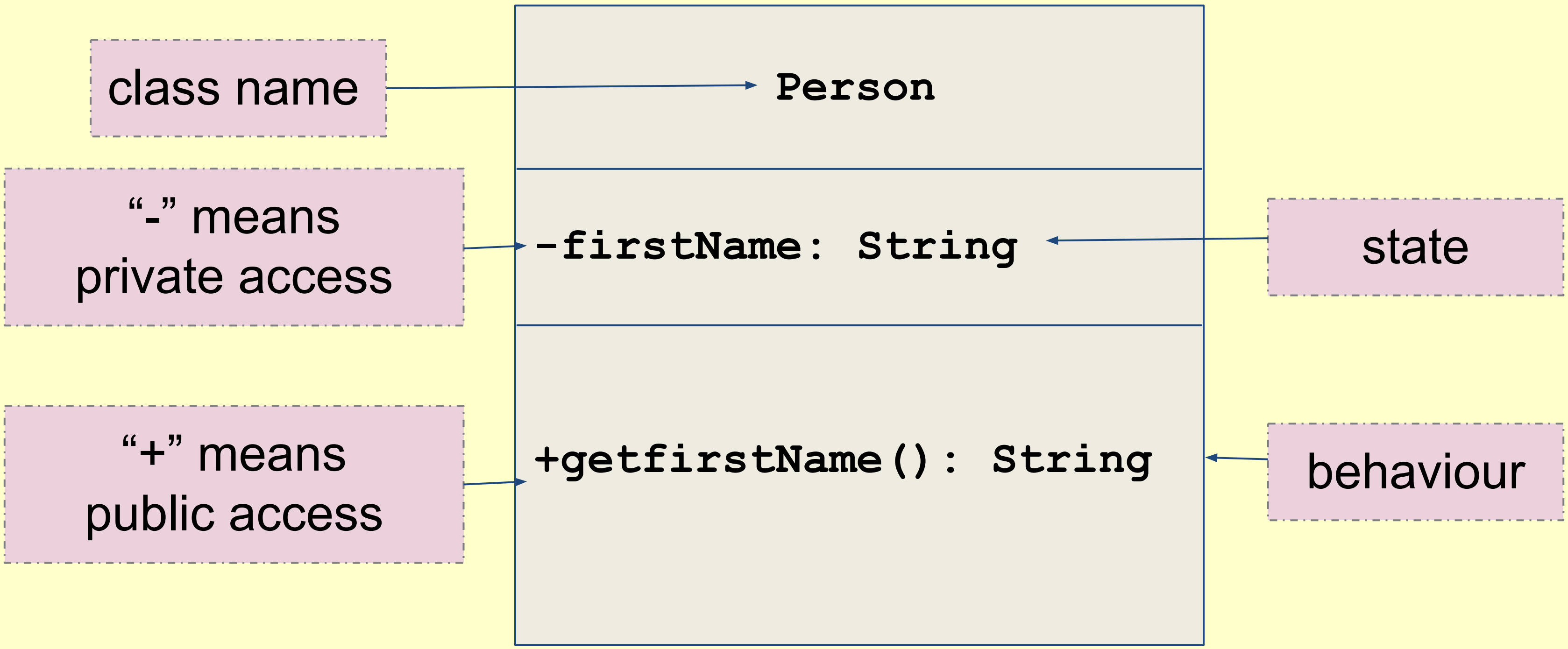
Encapsulation

- **Bundling** of **data** with the **methods** that operate on that data (restricting of direct access to some of an object's components)



- Hides the **implementation details** of a class
- Forces the user to use an **interface** to access data
- Makes the code more **maintainable**

UML - Graphical Class Representation



Declaring Constructors

- **Syntax:**

```
[<modifier>]<class_name>( <argument>*){
    <statement>*
}
```

```
public class Date {
  private int year, month, day;

  public Date( int y, int m, int d)    {
     if( verify(y, m, d) ){
        year = y; month = m; day = d;
     }
  }

  private boolean verify(int y, int m, int d){
    //...
  }
}
```

Constructors

- Role: **object initialization**
- **Name** of the constructor must be the same as that of class name.
- Must **not** have **return type**.
- Every class should have **at least one constructor**.
    - If you don't write constructor, compiler will generate the **default constructor**.
- Constructors are usually declared **public**.
    - Constructor can be declared as private → You can't use it outside the class.
- One class can have **more than one constructors.**
    - Constructor *overloading*.

The Default Constructors

- There is always **at least one constructor** in every class.
- If the programmer does not supply any constructors, the **default constructor** is generated by the compiler
  - The default constructor takes no argument
  - The default constructor's body is empty

default constructor

```
public class Date {
  private int year, month, day;

  public Date( ){
  }
}
```

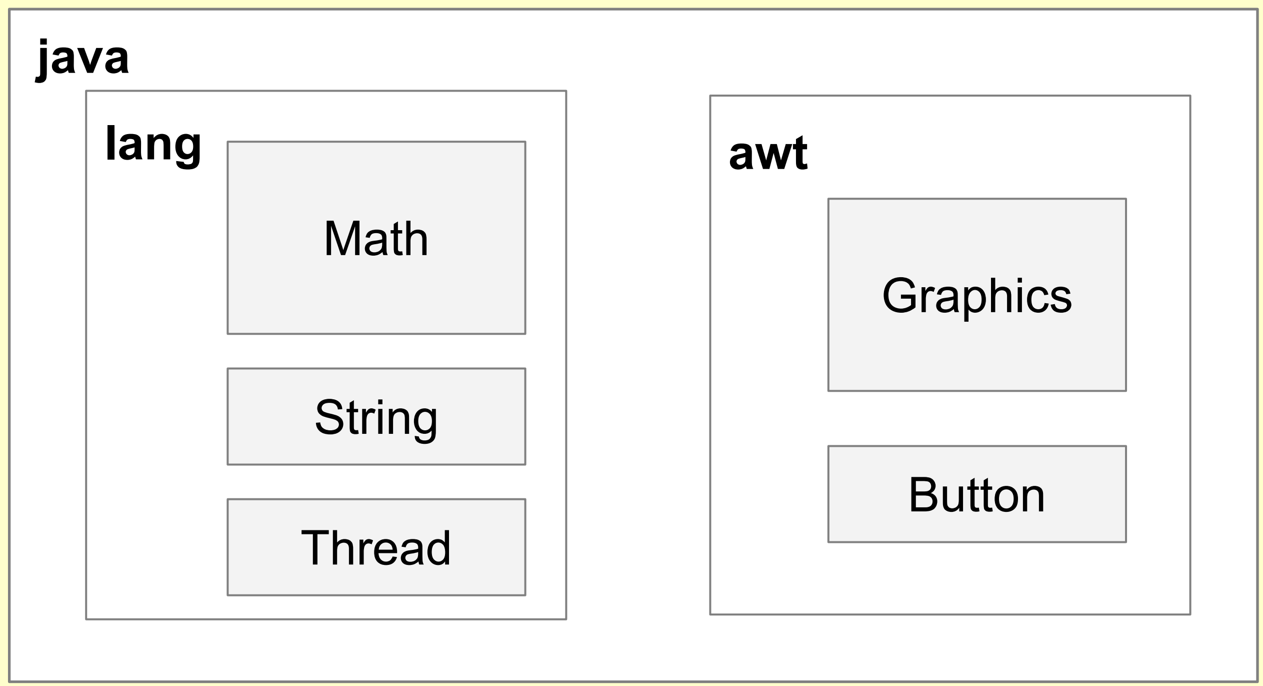
Objects

- Objects are **instances** of classes
- Are **allocated on the heap** by using the new operator
- Constructor is invoked automatically on the new object

```
Counter c = new Counter();

Date d1 = new Date( 2016, 9, 23);

Person p = new Person(“John”,”Smith”);
```

Packages

- Help manage large software systems
- Contain
  - Classes
  - Sub-packages



The `package` statement

- Syntax:

```
package <top_pkg_name>[.<sub_pkg_name>]*;
```

- Examples:

```
package java.lang;

public class  String{
    //...
}
```

- statement **at the beginning** of the source file
- only **one package declaration** per source file
- if **no package name** is declared → the class is placed into the **default package**

The `import` statement

- Syntax:

```
package <top_pkg_name>[.<sub_pkg_name>]*;
```

- Usage:

```
import <pkg_name>[.<sub_pkg_name>]*.*;
```

- Examples:

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

-precedes all class declarations
-tells the compiler **where to find classes**

Remember

- Class, encapsulation
- Class members:
  - attributes
  - methods
- Object, instance
- Constructor
- Package
- Import statement

Object-oriented programming

Types

- Primitive types
- Reference Type
- Parameter Passing
- The `this` reference
- Variables and Scope
- Casting

Java Types

- **Primitive (8)**
  - Logical: `boolean`
  - Textual: `char`
  - Integral: `byte, short, int, long`
  - Floating: `double, float`
- **Reference**
  - All others

Logical - `boolean`

- Characteristics:
  - Literals:
    - `true`
    - `false`
  - Examples:
    - `boolean cont = true;`
    - `boolean exists = false;`

Textual - `char`

- Characteristics:
  - Represents a 16-bit Unicode character
  - Literals are enclosed in single quotes (' ')
  - Examples:
    - `'a'` - the letter `a`
    - `'\t'` - the `TAB` character
    - `'\u0041'` - a specific Unicode character (`'A'`) represented by 4 hexadecimal digits

Integral – `byte`, `short`, `int`, and `long`

- Characteristics:
  - Use three forms:
    - Decimal: `67`
    - Octal: `0103 (1x8^2+0x8^1+3x8^0)`
    - Hexadecimal: `0x43`
  - Default type of literal is `int`.
  - Literals with the `L` or `l` suffix are of type `long`.

Integral – `byte`, `short`, `int`, and `long`

- Ranges:

Type	Length	Range
`byte`	1 byte	$-2^{7}..2^{7}-1$
`short`	2 byte	$-2^{15}..2^{15}-1$
`int`	4 byte	$-2^{31}..2^{31}-1$
`long`	8 byte	$-2^{63}..2^{63}-1$

Floating Point – `float` and `double`

- Characteristics:
  - **Size:**
    - `float  - 4 byte`
    - `double - 8 byte`
  - Decimal point
    - `9.65` (`double,` **default type**)
    - `9.65`**`f`** or `9.65`**`F`** (`float`)
    - `9.65`**`D`** or `9.65`**`d`** (`double`)
  - Exponential notation
    - `3.41E20` (`double`)

Java Reference Types

```java
public class  MyDate {
    private int day = 26;
    private int month = 9;
    private int year = 2016;

    public MyDate( int day, int month, int year){
    ...
    }
}
```

```java
MyDate date1 = new MyDate(20, 6, 2000);
```

Constructing and Initializing Objects

```
MyDate date1 = new MyDate(20, 6, 2000);
```

Constructing and Initializing Objects

```
MyDate date1 = new MyDate(20, 6, 2000);
```

```
new MyDate(20, 6, 2000);
```

1) Memory is allocated for the object
2) Explicit attribute initialization is performed
3) A constructor is executed
4) The **object reference** is returned by the `new` operator

Constructing and Initializing Objects

```
MyDate date1 = new MyDate(20, 6, 2000);
```

```
new MyDate(20, 6, 2000);
```

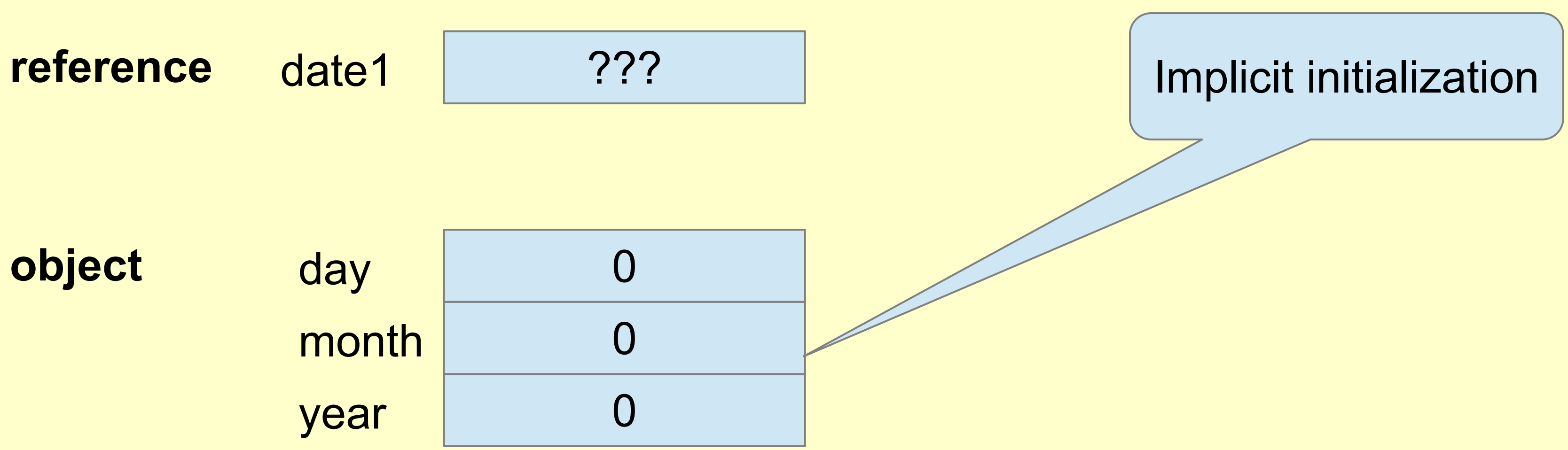
1) Memory is allocated for the object
2) Explicit attribute initialization is performed
3) A constructor is executed
4) The **object reference** is returned by the `new` operator

```
date1 = object reference
```

5) The reference is assigned to a variable

(1) Memory is allocated for the object

```
MyDate date1 = new MyDate(20, 6, 2000);
```



(2) Explicit Attribute Initialization

```
MyDate date1 = new MyDate(20, 6, 2000);
```

reference	date1	???
object	day	26
	month	9
	year	2016

```
public class  MyDate{
    private int day = 26;
    private int month = 9;
    private int year = 2016;

}
```

(3) Executing the constructor

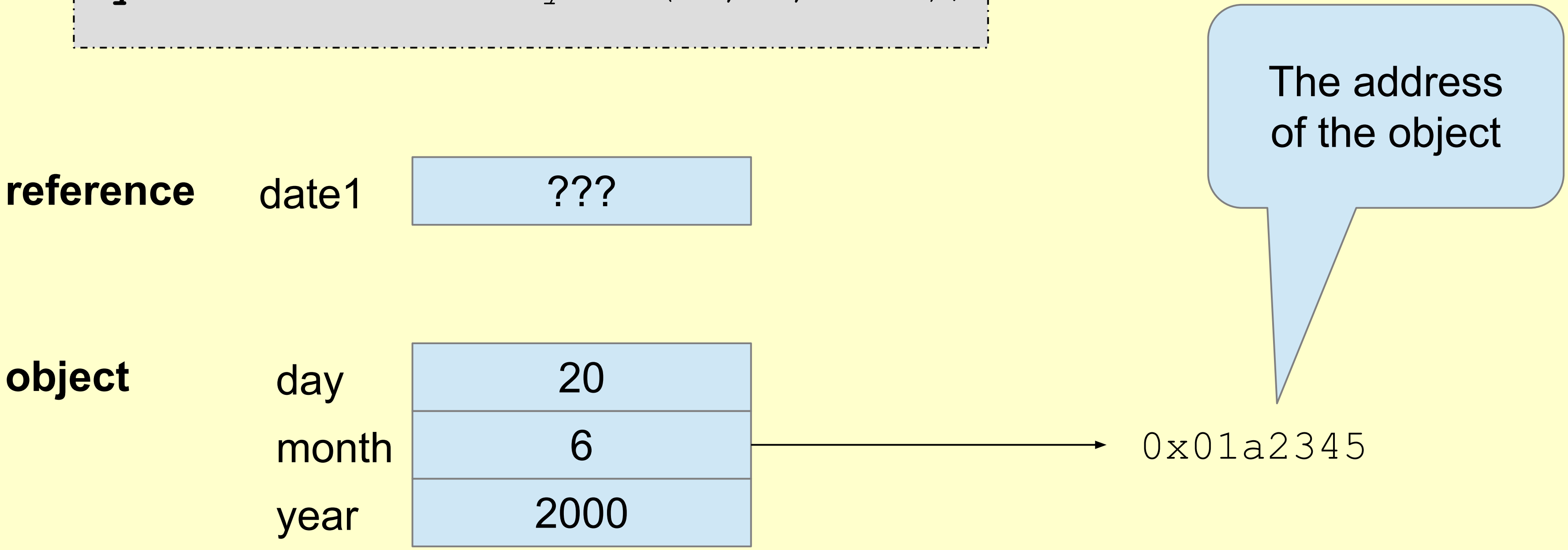
```
MyDate date1 = new MyDate(20, 6, 2000);
```

reference	date1	???
object	day	20
	month	6
	year	2000

```
public class  MyDate{
    private int day = 26;
    private int month = 9;
    private int year = 2016;

}
```

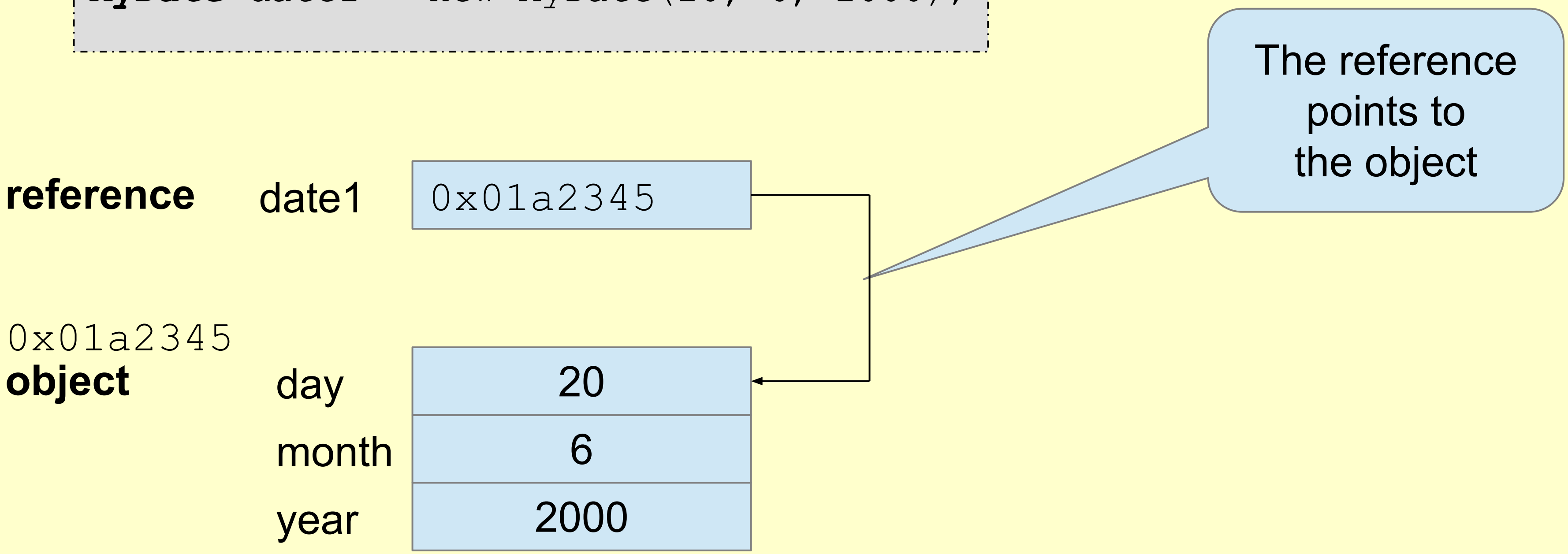
(4) The object reference is returned

```
MyDate date1 = new MyDate(20, 6, 2000);
```



(5) The reference is assigned to a variable

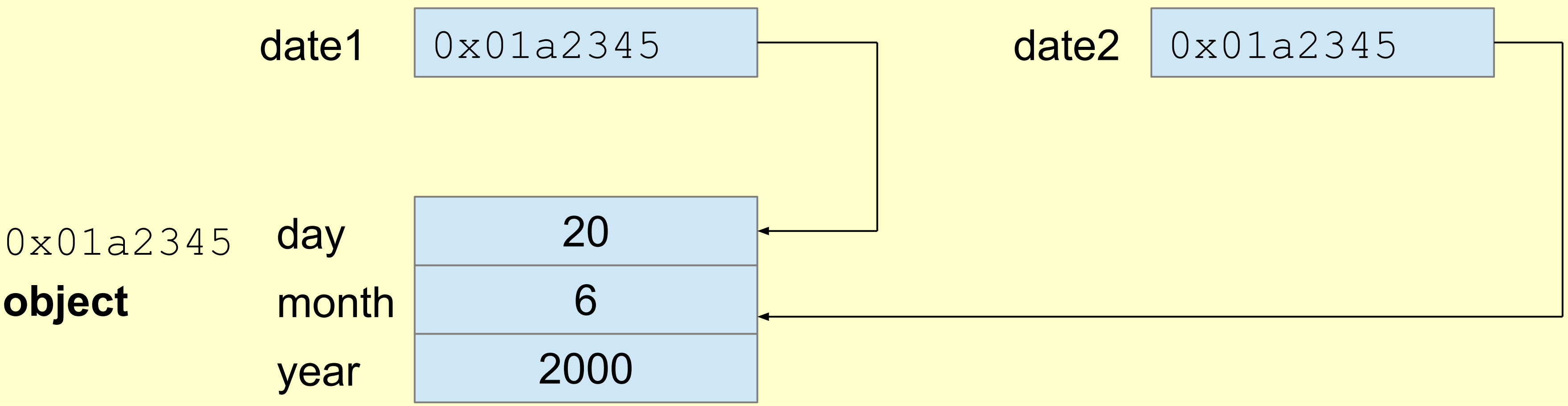
```
MyDate date1 = new MyDate(20, 6, 2000);
```



Assigning References

- Two variables refer to a single object

```
MyDate date1 = new MyDate(20, 6, 2000);
MyDate date2 = date1;
```



Parameter Passing

Pass-by-Value

```
public class  PassTest{
    public  void changePrimitive(int value){
        ++value;
    }

    public void changeReference(MyDate from, MyDate to){
        from = to;
    }

    public void changeObjectDay(MyDate date, int day){
        date.setDay( day );
    }
}
```

Parameter Passing

Pass-by-Value

```
PassTest pt = new PassTest();
int x = 100;
pt.changePrimitive( x );
System.out.println( x );

MyDate oneDate = new MyDate(3, 10, 2016);
MyDate anotherDate = new MyDate(3, 10, 2001);

pt.changeReference( oneDate, anotherDate );
System.out.println( oneDate.getYear() );

pt.changeObjectDay( oneDate, 12 );
System.out.println( oneDate.getDay() );
```

Output:
100
2016
12

The `this` Reference

- Usage:
  - To resolve **ambiguity** between *instance variables* and *parameters*
  - To **pass** the current **object as a parameter** to another method

The `this` Reference

```
public class  MyDate{
    private int day = 26;
    private int month = 9;
    private int year = 2016;
    public MyDate( int day, int month, int year){
        this.day   = day;
        this.month = month;
        this.year  = year;
    }
    public MyDate( MyDate date){
        this.day   = date.day;
        this.month = date.month;
        this.year  = date.year;
    }
    public MyDate creteNextDate(int moreDays){
        MyDate newDate = new MyDate(this);
        //... add moreDays
        return newDate;
    }
}
```

Java Coding Conventions

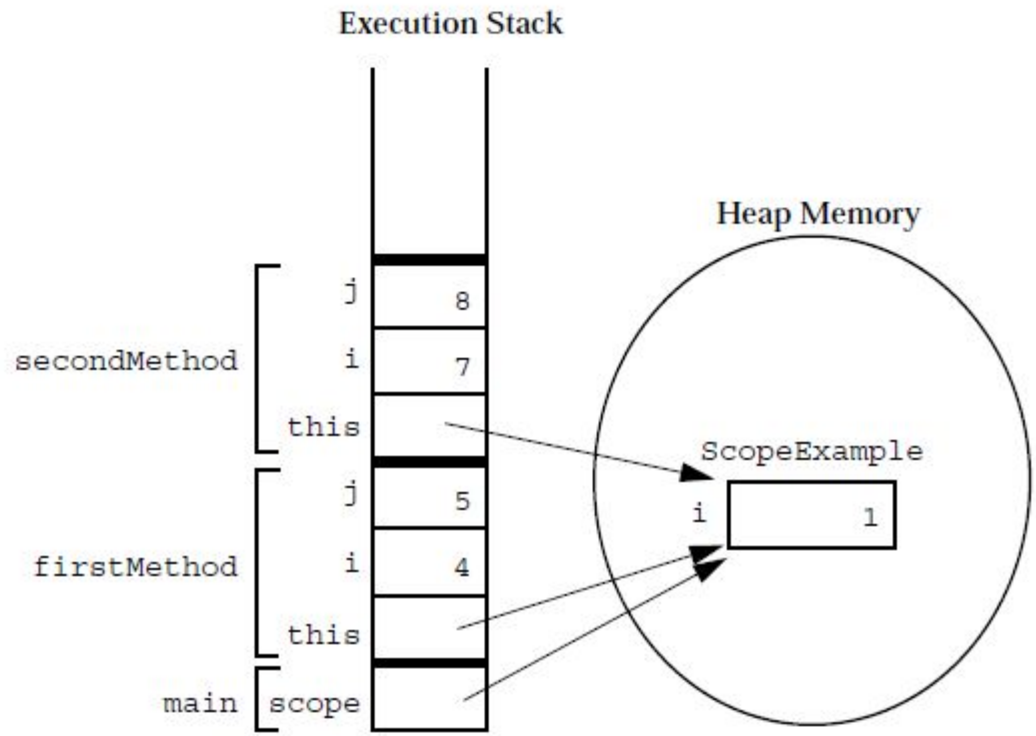
- Packages
  - `ro.sapientia.ms`
- Classes
  - `SavingsAccount`
- Methods
  - `getAmount()`
- Variables
  - `amount`
- Constants
  - `NUM_CLIENTS`

Variables and Scope

- Local variables are
  - Defined **inside a method**
  - Created when the **method is executed** and destroyed when the **method is exited**
  - **Not initialized automatically**
  - Created on the **execution stack**

Variable Scope Example

```
public class ScopeExample {
  private int i=1;

  public void firstMethod() {
    int i=4, j=5;

    this.i = i + j;
    secondMethod(7);
  }
  public void secondMethod(int i) {
    int j=8;
    this.i = i + j;
  }
}
```




```
public class TestScoping {
  public static void main(String[] args) {
    ScopeExample scope = new ScopeExample();

    scope.firstMethod();
  }
}
```


Copyright 2006 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision F.1

Default Initialization

- Default values for attributes:

Type	**Value**
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`char`	`'\u0000'`
`boolean`	`false`
`refrence`	`null`

Operators

- Logical operators
- Bitwise operators (~, ^, &, |, >>, >>>, <<)
- String concatenation (+)

String Types

- **`String`**
  - **Immutable –** once created can not be changed
  - Objects are stored in the **Constant String Pool**
- **`StringBuffer`**
  - **Mutable** – one can change the value of the object
  - **Thread-safe**
- **`StringBuilder`**
  - The same as `StringBuffer`
  - **Not thread-safe**

Object-oriented programming

Arrays

- Declaring arrays
- Creating arrays
- Arrays of primitive and reference type
- Initialization of elements
- Multidimensional arrays

Declaring Arrays

- What is an array?
  - **Group** of data objects of the **same type**
- Arrays of primitive types:

```
int t[];
int [] t;
```

- Arrays of reference types:

```
Point p[];
Point[] p;
```

Creating Arrays

Primitive Type

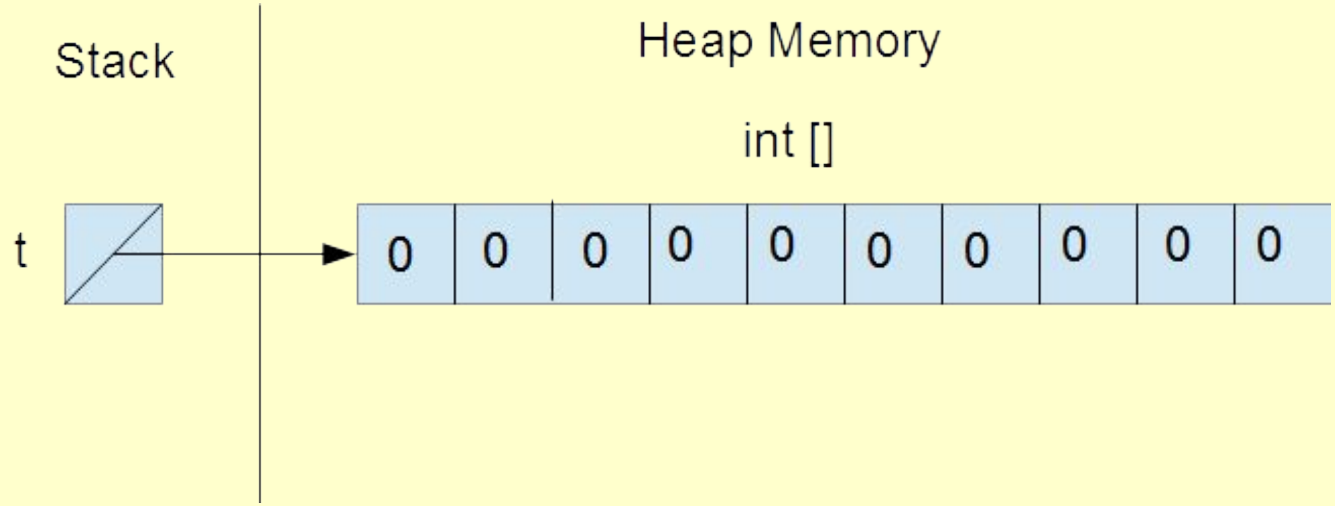
- Arrays are **objects** → are created with **new**
- Example:

```
//array declaration
int [] t;

//array creation
t = new int[10];

//print the array - enhanced for loop
for( int v: t ){
  System.out.println( v );
}
```

Creating Arrays

Primitive Type

```
//array declaration
int [] t;

//array creation
t = new int[10];
```



Creating Arrays

Reference Type

```
//array declaration
Point [] t;

//array creation - array of references!!!
t = new Point[3];
```

How many objects of type Point?

Creating Arrays

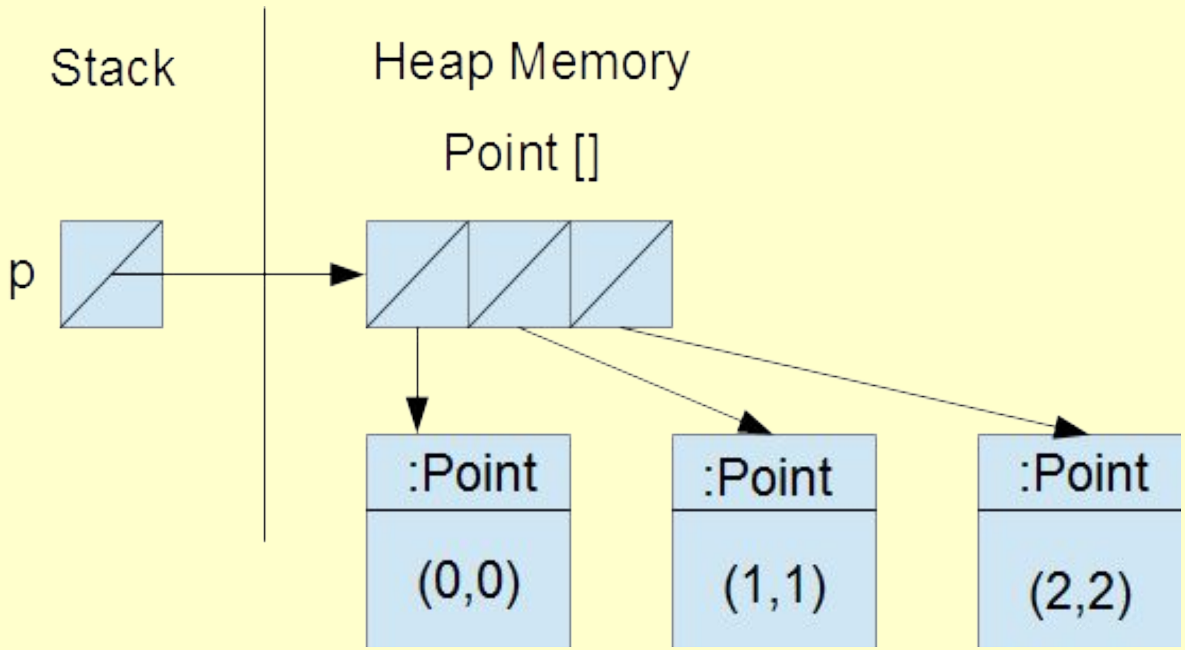
Reference Type

```
//array declaration
Point [] p;

//array creation - array of references!!!
p = new Point[3];

// Initializing references with objects
for( int i=0; i<3; ++i){
   p[i] = new Point(i, i);
}
```

How many objects of type Point?

Creating Arrays

Reference Type



Initializing Arrays

- Create an array with initial values

```
String names[] = {"Anna", "Krisztina", "Rebekka"};
Point points[] ={ new Point(0,0), new Point(1,1)};
```

Array Bounds

```
void printElements( int t[] ){

      for( int i=0; i < t.length; ++i){
         System.out.println( t[i] );
      }
}
```

Multidimensional Arrays

- **Rectangular** arrays:

```
int [][] array = new int[3][4];
```

- **Non-rectangular** arrays:

```
int [][] array;
array = new int[2][];
array[0] = new int[3];
array[1] = new int[5];
```

Remember

- Array **declaration** and **creation**
  - Array of primitives
  - Array of references
- Size of an array (public attribute: **length**)
- **Initial values** of array elements

Module 3
Static Members

Problems

- How can you create a **constant**?
- How can you declare **data** that is **shared by all instances of a given class**?
- How can you **prevent** a class from being **subclassed**?
- How can you **prevent** a method from being **overridden**?

Problem

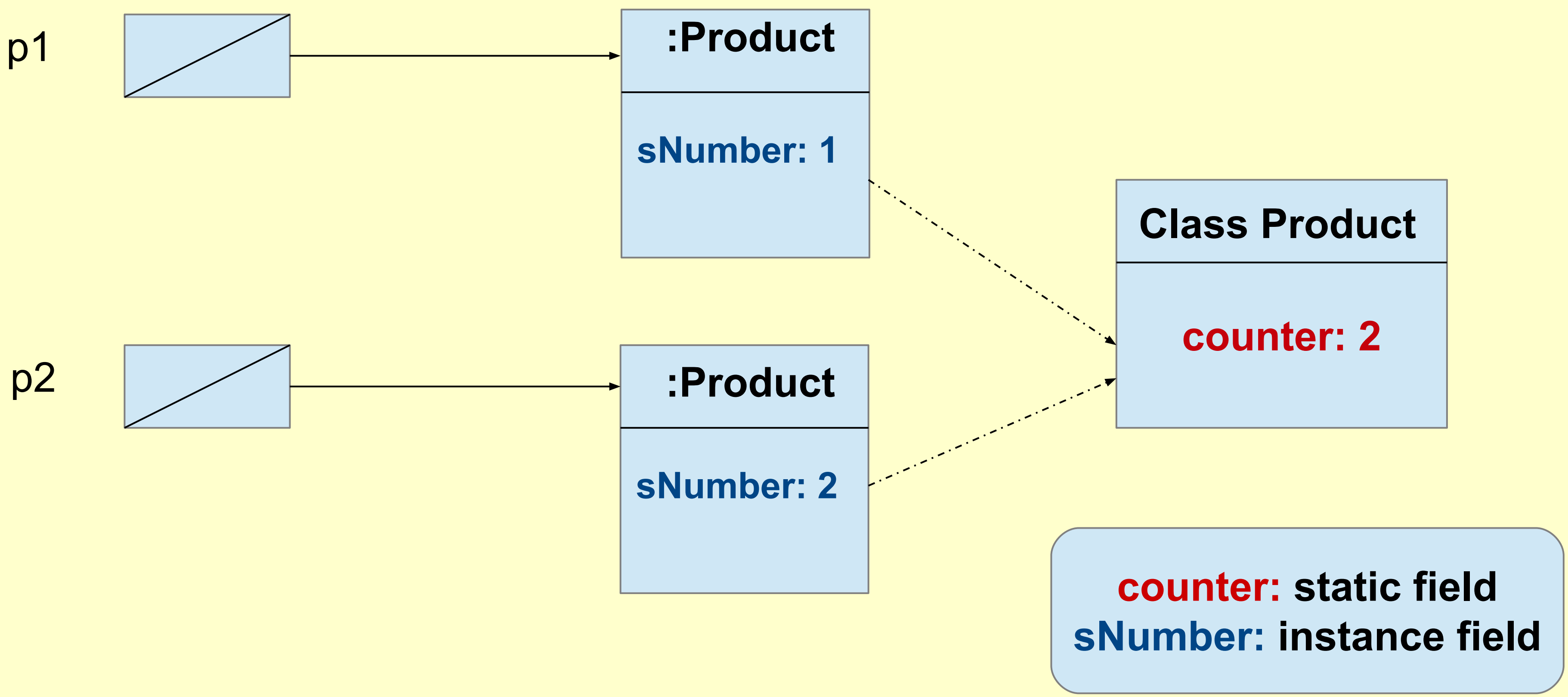
- Create a **`Product`** class which initializes each new instance with a **`serialNumber`** **`1,2, 3,…`**

Solution

```
public class Product{
    private int sNumber;
    public static int counter = 0;
    public Product() {
        counter++;
        sNumber = counter;
    }
}
```

Solution

```
Product p1 = new Product();
Product p2 = new Product();
```



What's wrong?

```
public class Product{
    private int sNumber;
    public static int counter = 0;
    public Product() {
        counter++;
        sNumber = counter;
    }
}


public class AnyClass{
    public void increment() {
        Product.counter++;
    }
}
```

It can be accessed from outside the class!

Better solution

```
public class Product{
    private int sNumber;

    private static int counter = 0;

    public static int getCounter(){
        return counter;
    }

    public Product() {
        counter++;
        sNumber = counter;
    }
}
```

Better solution

```
public class Product{
    private int sNumber;

    private static int counter = 0;

    public static int getCounter(){
        return counter;
    }

    public Product() {
        counter++;
        sNumber = counter;
    }
}
```

```
System.out.println(Product.getCounter());
Product p = new Product();
System.out.println(Product.getCounter());
```

Output?

Accessing static members

Recommended:

<class name>.<member_name>

Not recommended (but working):

<instance_reference>.<member_name>

```
System.out.println(Product.getCounter());
Product p = new Product();
System.out.println(p.getCounter());
```

Output?

Static Members

- Static data + static methods = static members
- Data are allocated at **class load time** → *can be used without instances*
- Instance methods **may use** static data. **Why?**
- Static methods **cannot use** instance data. **Why?**

The `InstanceCounter` class

```
public class InstanceCounter {
    private static int counter;

    public InstanceCounter(){
        ++counter;
    }

    public static int getCounter(){
        return counter;
    }
}
```

```
System.out.println( InstanceCounter.getCounter());

InstanceCounter ic = new InstanceCounter();
System.out.println( InstanceCounter.getCounter());
```

Output?

Singleton Design Pattern

```
public class Singleton {
    private static Singleton instance;

    private Singleton(){
    }

    public static Singleton getInstance(){
        if( instance == null ){
            instance = new Singleton();
        }
        return instance;
    }
}
```

Static Initializers

```
public class AClass{

  private static int counter;

  static {
     // e.g. read counter from a file
  }
}
```

The `final` Keyword

- **Class**
  - You cannot subclass a **`final`** class.
- **Method**
  - You cannot override a **`final`** method.
- **Variable**
  - A **`final`** variable is a constant.
  - You can set a **`final`** variable only once.
  - Assignment can occur independently of the declaration (*blank final variable*).

Blank Final Variables

```
public class Employee{
    private final long ID;

    public Employee(){
      ID = createID();
    }

    private long createID(){
       //return the generated ID
    }
    …
 }
```

Enumerations

```
public enum GestureType {
    UP,
    RIGHT,
    DOWN,
    LEFT
}

--------------------------------------------------

for(GestureType type: GestureType.values()){
    System.out.println( type );
}

OUTPUT:
     UP
     RIGHT
     DOWN
     LEFT
```

Enumerations

```java
public enum GestureType {
    UP (0, "fel"),
    RIGHT (1, "jobb"),
    DOWN (2, "le"),
    LEFT (3, "bal");

    GestureType( int value, String name ){
        this.value = value;
        this.name = name;
    }

    public int getValue(){
        return value;
    }

    public String getName(){
        return name;
    }

    private int value;
    private String name;
}
```

Enumerations

```
for(GestureType type: GestureType.values()){
    System.out.println(type.name()+", "+
                       type.getName()+", "+ type.getValue());
}
```

Output

```
UP, fel, 0
RIGHT, jobb, 1
DOWN, le, 2
LEFT, bal, 3
```

REMEMBER

- **Constant instance data**
  - belongs to the **instance**
- **Static data**
  - belongs to the **class**
- **Constant static data**
  - belongs to the **class**

REMEMBER

CONSTANT INSTANCE DATA

final

```
public class Product{
      private final int ID;
}
```

REMEMBER

STATIC DATA

static

```
public class Product{
    private final int ID;
    private static counter;
    public Product(){
        ID = ++counter;
    }
}
```

REMEMBER

CONSTANT STATIC DATA

static final

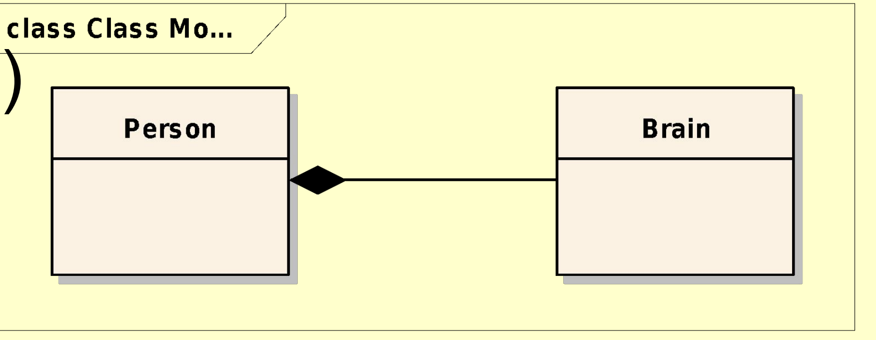
```
public class Product{
    private final int ID;
    private static counter;
    private static final String name = "PRODUCT";
    public Product(){
        ID = ++counter;
    }
    public String getIDStr(){
        return name+ID;
    }
}
```

Module 4
Relationships between classes

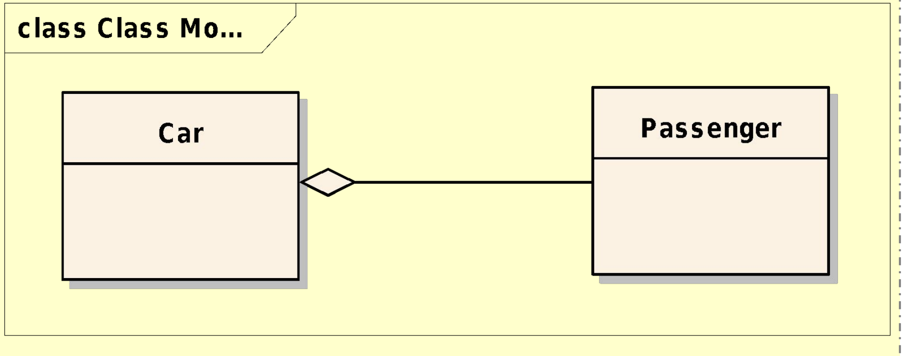
Object-oriented programming

Relationships between classes

- **Association** (containment)
  - **Strong – Composition**

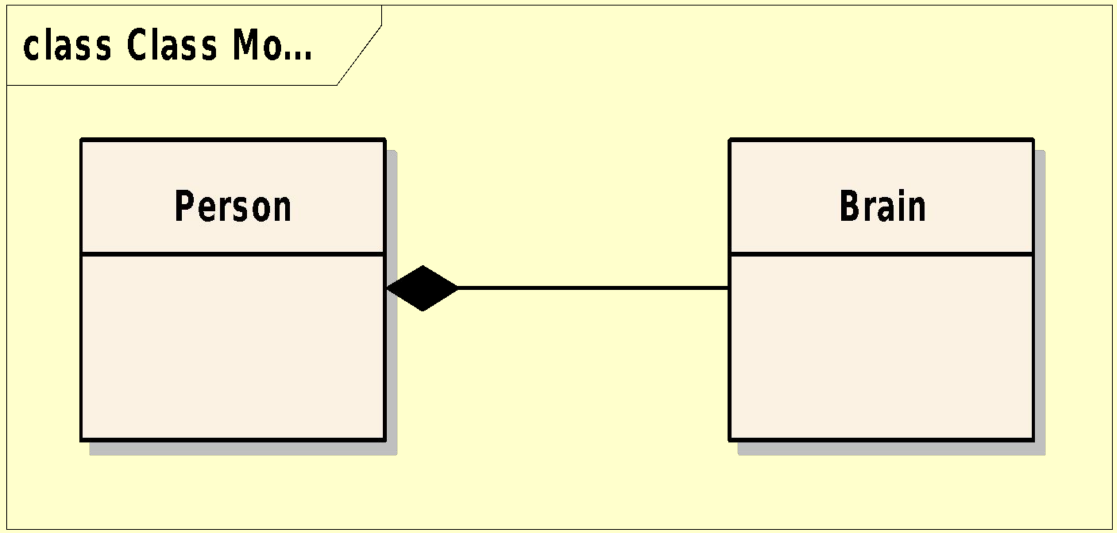


  - **Weak – Aggregation**



Relationships between classes

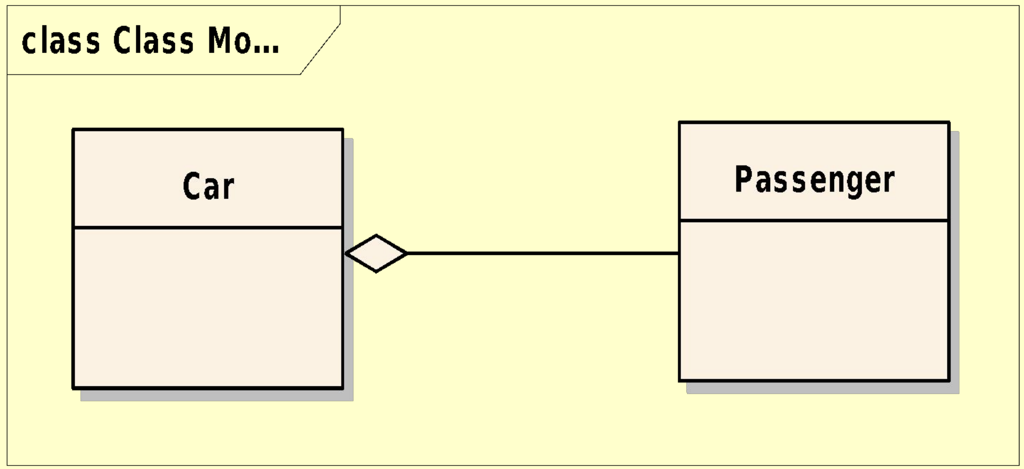
Composition



- Strong type of association
- Full ownership

Relationships between classes

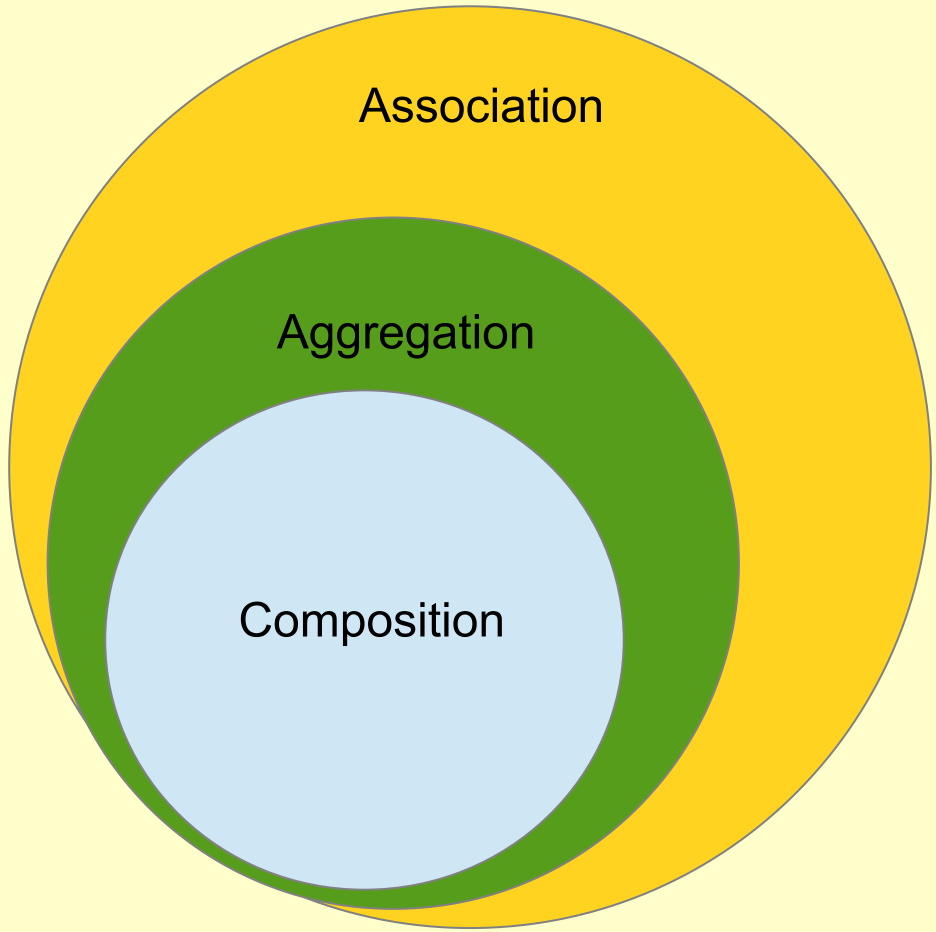
Aggregation



- Weak type of association
- Partial ownership

Relationships between classes

Association – Aggregation - Composition



Relationships between classes

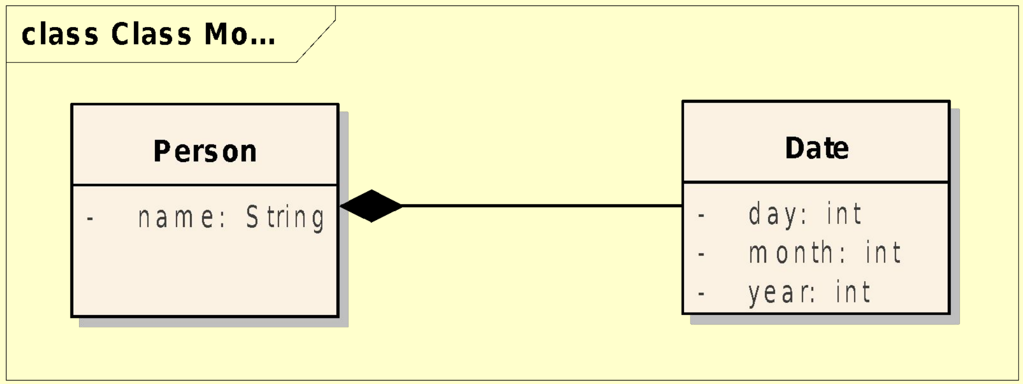
Implementing Associations (1)

```
public  class Brain{
    //...

}
```

```
public class Person{
   private Brain brain;
   //...
}
```

Relationships between classes

Implementing Associations (2)



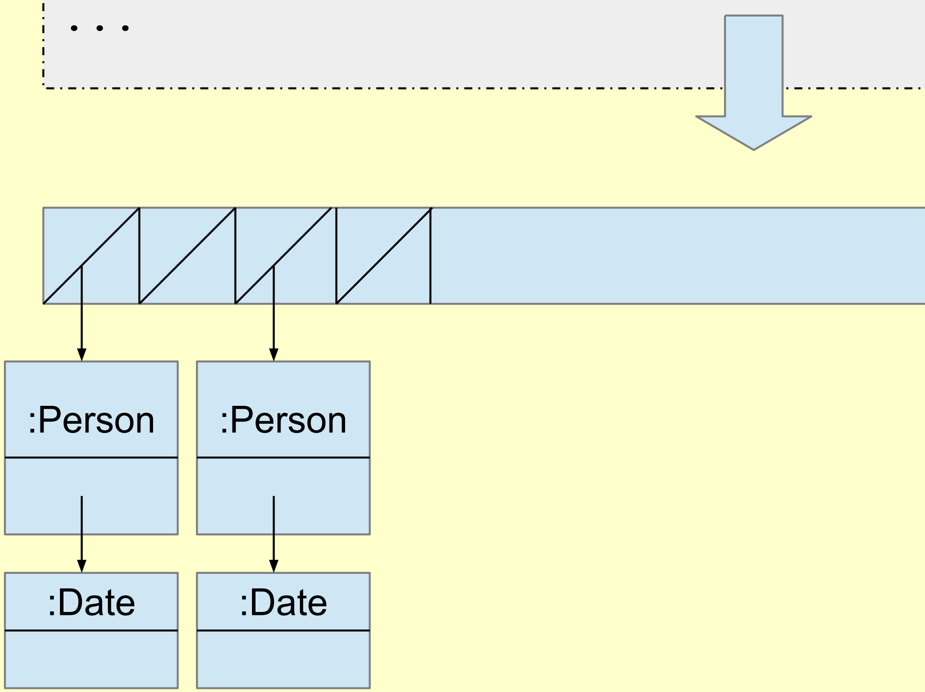

```
public  class Date{
    private int day;
    private int month;
    private int year;
    //...
}
```

```
public class Person{
    private String name;
    private Date birthDate;

    public Person(String name,
                  Date birthDate){
        this.name = name;
        this.birthDate = birthDate; }
    //...
}
```

Relationships between classes

Implementing Associations (3)

```
Benedek Istvan, 1990, 1, 12
Burjan  Maria, 1968, 4, 15
Dobos Hajnalka Evelin, 1986, 3, 17
...
```

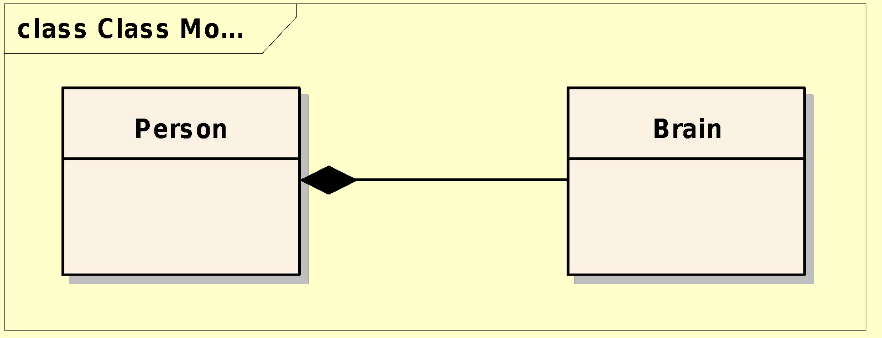


Write a program which reads the data of several persons and constructs an array of Persons.

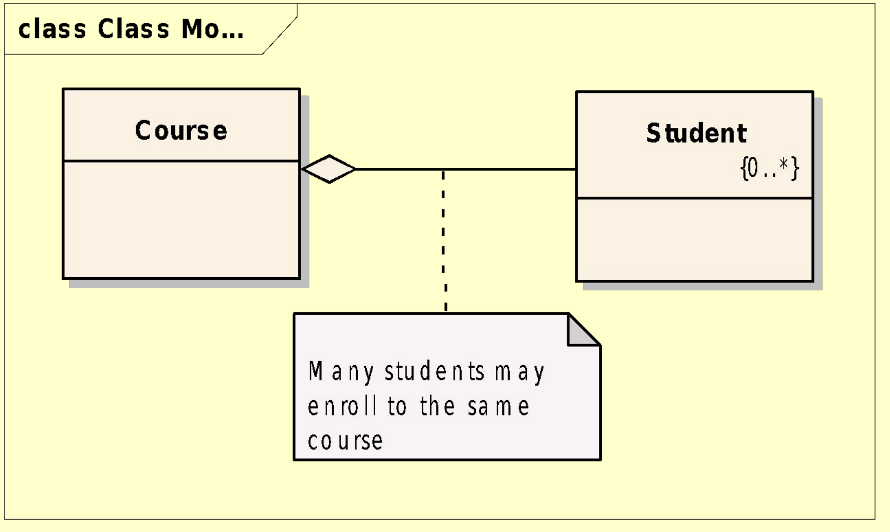
Relationships between classes

Relationship cardinality

- *One-to-one*



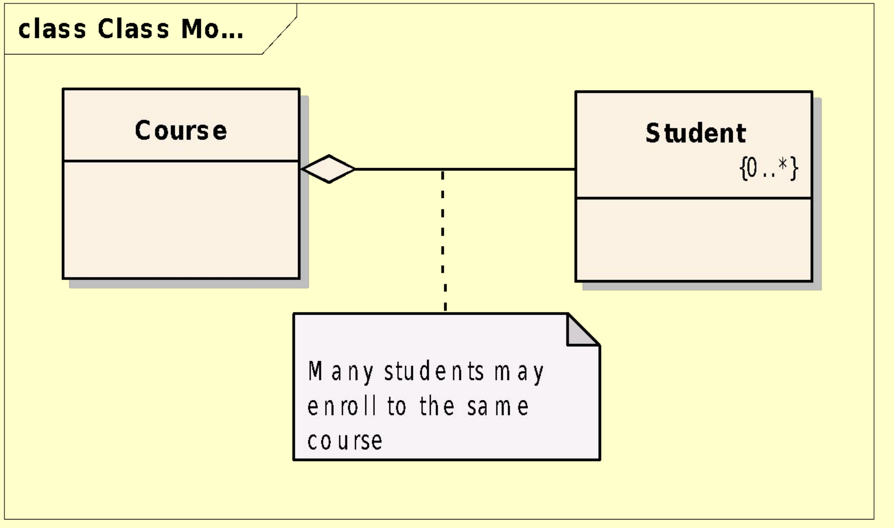
- *One-to-many*



Relationships between classes

Implementing *one-to-many* relationship (1)

```
public  class Student{
    private final long ID;
    private String firstname;
    private String lastname;
    //...
}
```




```
public  class Course{
    private final long ID;
    private String name;
    public static final int MAX_STUDENTS=100;
    private Student[] enrolledStudents;
    private int numStudents;

    //...
}
```

Relationships between classes

Implementing *one-to-many* relationship (2)

```
public  class Course{
    private final long ID;
    private String name;
    public static final int MAX_STUDENTS = 100;
    private Student[] enrolledStudents;
    private int numStudents;

    public Course( long ID, String name ){
        this.ID = ID;
        this.name = name;
        enrolledStudents = new Student[ MAX_STUDENTS ];
    }

    public void enrollStudent( Student student ){
        enrolledStudents[ numStudents ] = student;
        ++numStudents;
    }
    //...
}
```

Relationships between classes

Implementing *one-to-many* relationship (3)

```
public  class Course{
    private final long ID;
    private String name;

    private ArrayList<Student> enrolledStudents;

    public Course( long ID, String name ){
        this.ID = ID;
        this.name = name;
        enrolledStudents = new ArrayList<Student>();
    }

    public void enrollStudent( Student student ){
        enrolledStudents.add(student);
    }


    //...
}
```

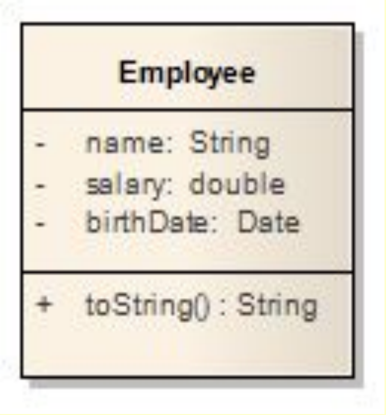
Module 5

Inheritance, Polymorphism

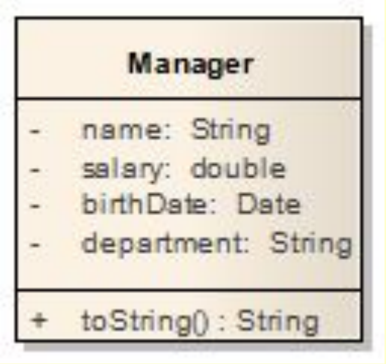
Outline

- Inheritance
    - Parent class
    - Subclass, Child class
- Polymorphism
    - Overriding methods
    - Overloading methods
    - The `instanceof` operator
    - Heterogeneous collections

Problem: *repetition in implementations*

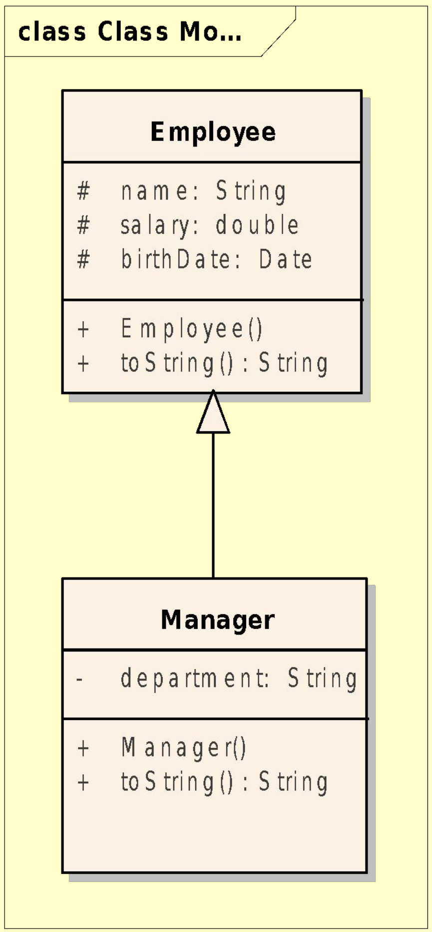



```java
public  class Employee{
    private String name;
    private double salary;
    private Date birthDate;

    public String toString(){
       //...
    }
}
```




```java
public  class Manager{
    private String name;
    private double salary;
    private Date birthDate;
    private String department;

    public String toString(){
       //...
    }
}
```

Solution: *inheritance*




```
public  class Employee{
     protected String name;
     protected double salary;
     protected Date birthDate;
     public Employee( … ){
          // …
     }
     public String toString(){
        //...
     }
}
```

```
public  class Manager extends Employee{
     private String department;

     public Manager( … ){
          // …
     }
     public String toString(){
        // …
     }
}
```

Inheritance - syntax

```
<modifier> class <name> extends <superclass>{
    <declaration*>
}
```

```
public  class Manager extends Employee{

}
```

The subclass

- **Inherits** the **data** and **methods** of the **parent class**
- **Does not inherit the constructors** of the **parent class**
- Opportunities:

  1) add new data
  2) add new methods
  3) override inherited methods (polymorphism)

The subclass

- Opportunities:
  1) add new data → `department`
  2) add new methods → e.g. `getDepartment()`
  3) override inherited methods → `toString()`

Invoking Parent Class Constructors

```
public  class Employee{
     protected String name;
     protected double salary;
     protected Date birthDate;
     public Employee( String name, double salary, Date birthDate){
          this.name = name;
          this.salary = salary;
          this.birthDate = birthDate;
     }
     //...
}
```

```
public  class Manager extends Employee{
     private String department;
     public Manager( String name, double salary, Date birthDate,
                                                String department){
     super(name, salary, bityhDate);
          this.department = department;
     }
     //...
}
```

Access Control

Modifier	Same Class	Same Package	Subclass	Universe
`private`	`Yes`			
default!	`Yes`	`Yes`		
`protected`	`Yes`	`Yes`	`Yes`	
`public`	`Yes`	`Yes`	`Yes`	`Yes`

Polymorphism - Overriding Methods

- A subclass can modify the **behavior** inherited from a parent class
- A subclass can create a method with different functionality than the parent's method but with the:
  - same **name**
  - same **argument list**
  - almost the same **return type**

(can be a subclass of the overriden return type)

Overriding Methods

```
public  class Employee{
     protected String name;
     protected double salary;
     protected Date birthDate;
     public Employee( … ){
          // …
     }
     public String toString(){
          return "Name: "+name+" Salary: "+salary+" B. Date:"+birthDate;
     }
}
```

```
public  class Manager extends Employee{
     private String department;
     public Manager( … ){
          // …
     }
     @Override
     public String toString(){
          return "Name: "+name+" Salary: "+salary+" B. Date:"+birthDate
                         +" department: "+department;
     }
}
```

Invoking Overridden Methods

```
public  class Employee{
     protected String name;
     protected double salary;
     protected Date birthDate;
     public Employee( … ){
          // …
     }
     public String toString(){
          return “Name: “+name+” Salary: ”+salary+” B. Date:”+birthDate;
     }
}
```

```
public  class Manager extends Employee{
     private String department;
     public Manager( … ){
          // …
     }
     public String toString(){
          return super.toString() + ” department: ”+department;
     }
}
```

Overridden Methods Cannot Be Less Accessible

```java
public  class Parent{
   public void foo(){}
}

public class Child extends Parent{
   private void foo(){} //illegal
}
```

Overriding Methods

- *Polymorphism:* the ability to have many different forms

```
Employee e = new Employee(...);
System.out.println( e.toString() );

e = new Manager(...); //Correct
System.out.println( e.toString() );
```

Which `toString()` is invoked?

Polymorphic Arguments

```
public String createMessage( Employee e ){
   return “Hello, ”+e.getName();
}

//...
Employee e1 = new Employee(“Endre”,2000,new Date(20,8, 1986));
Manager m1 = new Manager(“Johann”,3000,
                          new Date(15, 9, 1990),”Sales”);

//...
System.out.println( createMessage( e1 ) );
System.out.println( createMessage( m1 ) );
```

Liskov Substitution!

Heterogeneous Arrays

```
Employee emps[] = new Employee[ 100 ];
emps[ 0 ] = new Employee();
emps[ 1 ] = new Manager();
emps[ 2 ] = new Employee();
// …

// print employees
for( Employee e: emps ){
   System.out.println( e.toString() );
}

// count managers
int counter = 0;
for( Employee e: emps ){
   if( e instanceof Manager ){
       ++counter;
   }
}
```

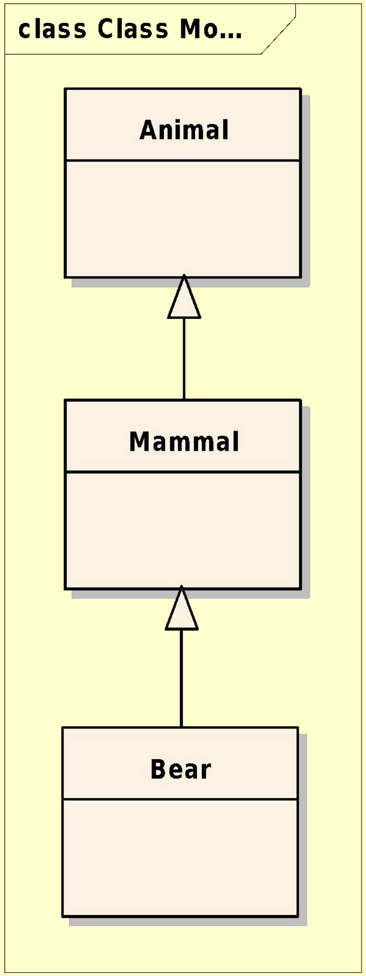
Static vs. Dynamic type of a reference

```
// static (compile time) type is: Employee
Employee e;

// dynamic (run time) type is: Employee
e = new Employee();

// dynamic (run time) type is: Manager
e = new Manager();
```

Static vs. Dynamic type of a reference

```
Employee e = new Manager("Johann",3000,
                         new Date(10,9,1980),"sales");
System.out.println( e.getDepartment() );// ERROR

//Solution
System.out.println( ((Manager) e).getDepartment() );// CORRECT

//Better Solution
if( e instanceof Manager ){
    System.out.println( ((Manager) e).getDepartment() );
}
```

The `instanceof` Operator




```
Animal a = new Bear();

//expressions
a instanceof Animal  → true
a instanceof Mammal → true
a instanceof Bear → true
a instanceof Date → false
```

Polymorphism

Overloading Methods

- **Polymorphism:** *the ability to have many different forms*
- Methods overloading:
  - methods having the **same name**,
  - argument list **must** differ,
  - return types **can be** different.
- Example:

```
public void println(int i)
public void println(float f)
public void println(String s)
```

Polymorphism

Overloading Constructors

```
public  class Employee{
     protected String name;
     protected double salary;
     protected Date birthDate;
     public Employee( String name, double salary, Date birthDate){
          this.name = name;
          this.salary = salary;
          this.birthDate = birthDate;
     }
     public Employee( String name, double salary){
          this(name, salary, null);
     }
     public Employee( String name, Date birthDate){
          this(name, 1000, birthDate);
     }
     //...
}
```

Polymorphism

The ability to have many different forms

- Methods **overloading**
  - **same name**, *different signature*
  - e.g. a class having multiple constructor
  - **compile-time** polymorphism (**static polymorphism**)
- Methods **overriding**
  - **same name**, *same signature*
  - e.g. `toString()`
  - **run-time** polymorphism (**dynamic polymorphism**)

Remember

- Inheritance
  - Subclass opportunities
- Polymorphism
  - *Overriding* methods
  - *Overloading* methods
  - *Polymorphic* argument
  - *Heterogeneous* collections
  - Static vs. dynamic type
  - The `instanceof` operator

Inheritance and Polymorphism

Methods Common to All Objects

- The `equals` method
- The `toString` method
- The `clone` method

Inheritance and Polymorphism

Methods Common to All Objects

- `Object` is a concrete class with non final methods:
  - `equals`
  - `toString`
  - `clone, …`
- **It is designed for extension!**
- Its methods have explicit *general contracts*

The `equals` method

In class Object `equals` tests object identity

```
MyDate s1  = new MyDate(20, 10, 2016);
MyDate s2  = new MyDate(20, 10, 2016);
System.out.println( s1.equals(s2));
s1 = s2;
System.out.println( s1.equals(s2));
```

Output?

An `equals` example

```
public class MyDate {
    private int day;
    private int month;
    private int year;

    public boolean equals(Object o) {
        boolean result = false;
        if ( (o != null) && (o instanceof MyDate) ) {
            MyDate d = (MyDate) o;
            if ((day == d.day) &&
                (month == d.month) &&
                (year == d.year)) {
                 result = true;
            }
        }
        return result;
    }

}
```

Another `equals` example

```
public class MyDate {
   private int day;
   private int month;
   private int year;

   @Override
   public boolean equals(Object o) {
       if (this == o) return true;
       if (o == null || this.getClass() != o.getClass()) return false;
       MyDate myDate = (MyDate) o;
       return day == myDate.day && month == myDate.month &&
              year == myDate.year;
   }
}
```

The `equals` method

In class MyDate `equals` tests object logical equality

```
MyDate s1  = new MyDate(20, 10, 2016);
MyDate s2  = new MyDate(20, 10, 2016);
System.out.println( s1.equals(s2));
s1 = s2;
System.out.println( s1.equals(s2));
```

Output?

The `equals` method implements an equivalence relation

- **Reflexive**

  `x.equals(x)`:**`true`**

- **Symmetric**

  `x.equals(y)`:**`true`** $\leftrightarrow$ `y.equals(x)`:**`true`**

- **Transitive**

`x.equals(y)`:**`true and`** `y.equals(z)`:**`true`** $\rightarrow$

`x.equals(z)`:**`true`**

The `toString` method

- Characteristics:
  - Converts an object to a `String`
  - Override this method to provide information about a user-defined object in **readable format**

Wrapper Classes

Primitive Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper Classes

Boxing and Unboxing

```
int i = 420;

Integer anInt = i; // boxing - creates new Integer(i);

int j = anInt; // unboxing - calls anInt.intValue();
```

Wrapper Classes

Warning! Performance loss!

```
public static void main(String[] args) {
   Long sum = 0L;
   for (long i = 0; i < Integer.MAX_VALUE; i++) {
      sum += i;
   }
   System.out.println(sum);
}
```

Too slow!!!

Module 6
Interfaces and Abstract Classes

Outline

- Interfaces
- Interfaces (since Java 8)
- Abstract classes
- Sorting
    - `Comparable` interface
    - `Comparator` interface

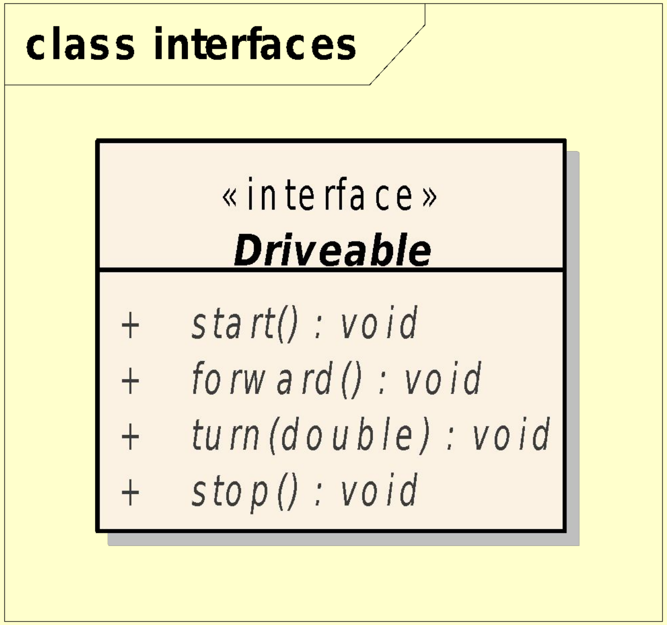
Interfaces

- Properties
  - Define **types**
  - Declare a **set of methods** (*no implementation!*)

    – ADT – Abstract Data Type
  - Will be **implemented** by classes

The `Driveable` Interface

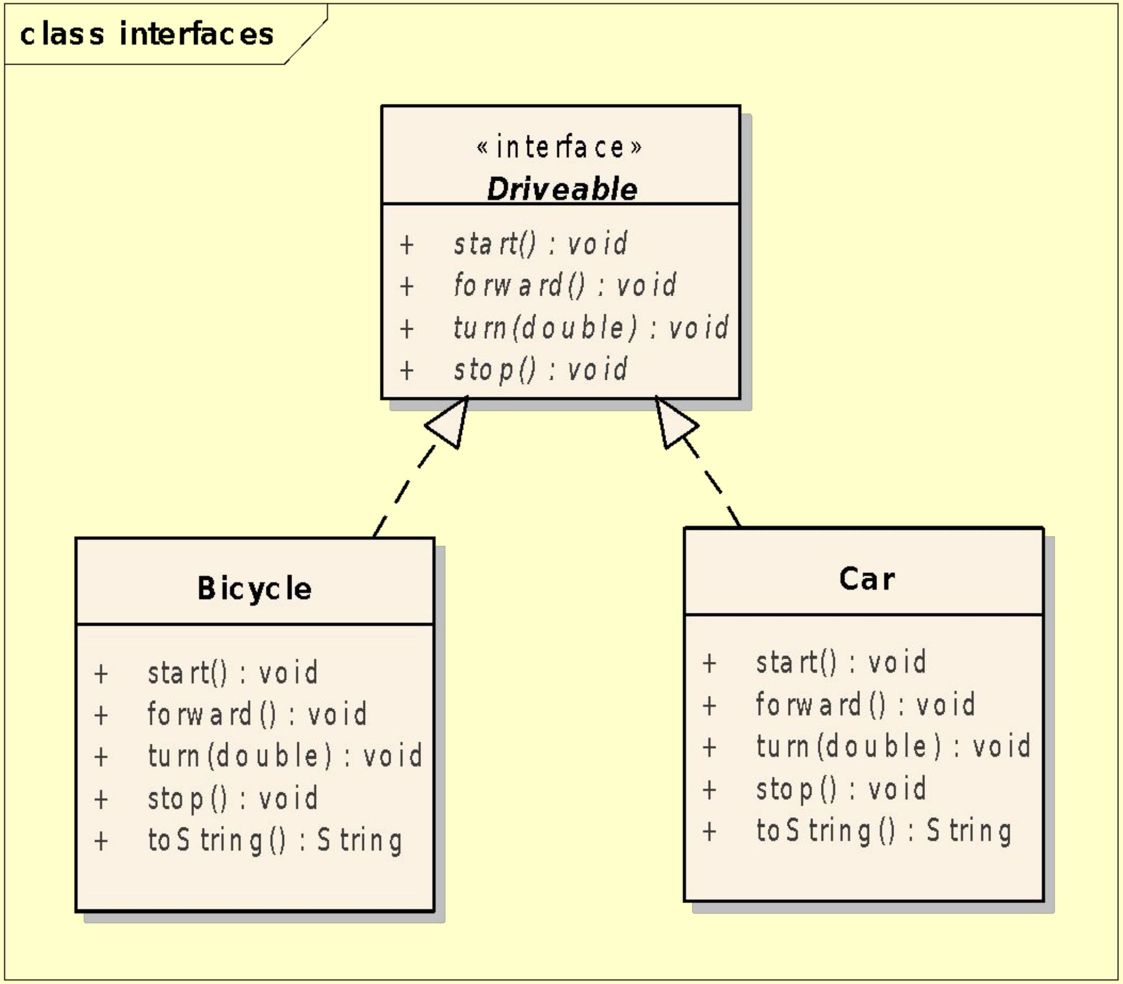
```
public interface Driveable{
      public void start();
      public void forward();
      public void turn( double angle);
      public void stop();
 }
```



Implementing Interfaces

```
public class Bicycle implements Driveable{
    @Override
    public void start() {
        System.out.println("The bicycle has been started");
    }

    @Override
    public void forward() {
        System.out.println("The bicycle moves forward");
    }

    @Override
    public void turn( double angle) {
        System.out.println("The bicycle turns "+angle+
                                            " clockwise");
    }

    @Override
    public void stop() {
        System.out.println("The bicycle has been stopped");
    }
 }
```

Implementing the `Driveable` Interface



Interfaces

- The interface contains **method declarations** and may contain constants
- All the methods are **public** (even if the modifier is missing)
- Interfaces are **pure abstract classes** → cannot be instantiated
- The implementer classes should **implement all the methods** declared in the interface
- A **class** can **extend a single class** but may **implement any number of interfaces**

Iterator interface

```
List<String> l1 = new ArrayList<>();
l1.add("Welcome");
l1.add("to");
l1.add("Java");

Iterator<String> it = l1.iterator();
while( it.hasNext() ){
   System.out.print( it.next() + " ");
}
-----------------------------------

for(String str: l1){
   System.out.print( str + " ");
}
```

Q & A

Select the correct statements!

a) `Driveable a;`

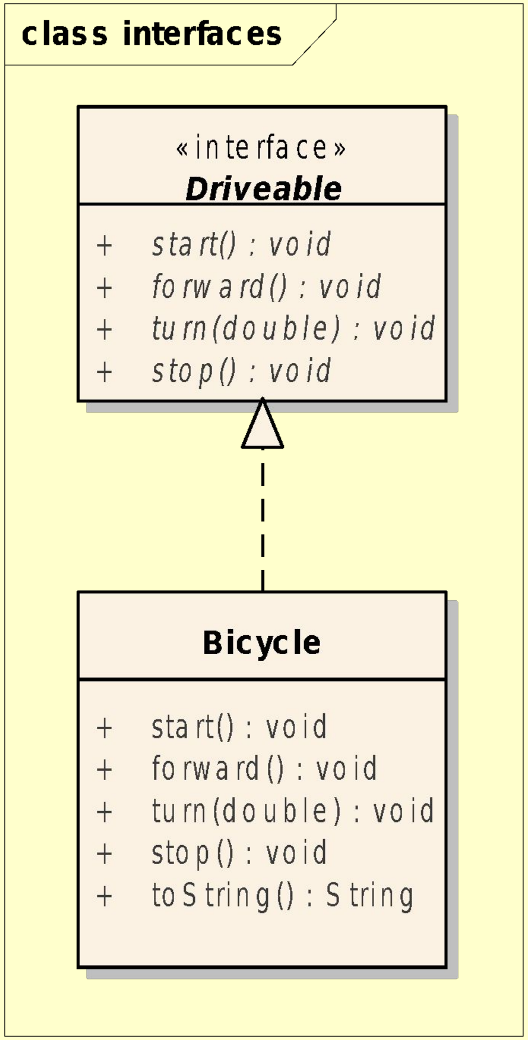
b) `Driveable a = new Driveable();`

c) `Driveable t[] = new Driveable[ 3 ];`

d) `public void drive( Driveable d );`

Interfaces vs. Classes

- **Interface:**
  - User-defined type
  - Set of methods
  - No implementations provided
  - Cannot be instantiated
- **Class:**
  - User-defined type
  - Set of data and methods
  - All the methods are implemented
  - Can be instantiated



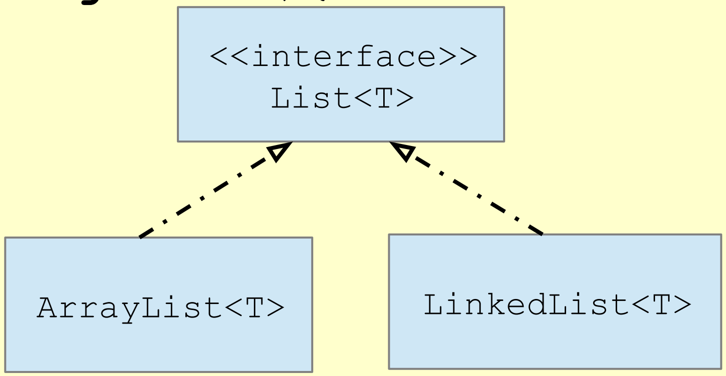
Polymorphic Argument

```
public class Utils{

   public static void moveMe(Driveable v){
      v.start();
      for( int i=0; i<12; ++i){
        v.turn(15);
       }
       v.stop();
   }
}
Utils.moveMe( new Bicycle() );
Utils.moveMe( new Car() );
```

What am I doing?

Polymorphic Argument

```
public class Utils{
   public static void printIt(List<String> list){
      for( String s: list ){
         System.out.println( s );
      }
   }
}
```




```
List<String> l1 = new ArrayList<>();
// add elements to l1
Utils.printIt(l1);
List<String> l2 = new LinkedList<>();
// add elements to l2
Utils.printIt(l2);
```

Interfaces Java 8

- Java Interface **Default** Method
- Java Interface **Static** method

Java Interface **Default** Method

```
public interface Animal{
   // Abstract method
   void eat();
   // Implemented method
   default void log( String str ){
      System.out.println(
           "Animal log: "+str);
   }
}
```

Java Interface **Default** Method

```
public class Bear implements Animal{
   // Mandatory!!!
   void eat(){
      System.out.println("Bear eats");
   }
   // It is not mandatory to provide
   // implementation for the log method
}
```

Java Interface **Static** Method

```
public interface MatrixOperations{

    static Matrix add(Matrix a, Matrix b){
        //...
    }

}
```

Java Interface **Static** Method

```
public interface MatrixOperations{

   static Matrix add(Matrix a, Matrix b){
      //...
   }

}
```

Java Interface **Static** Method

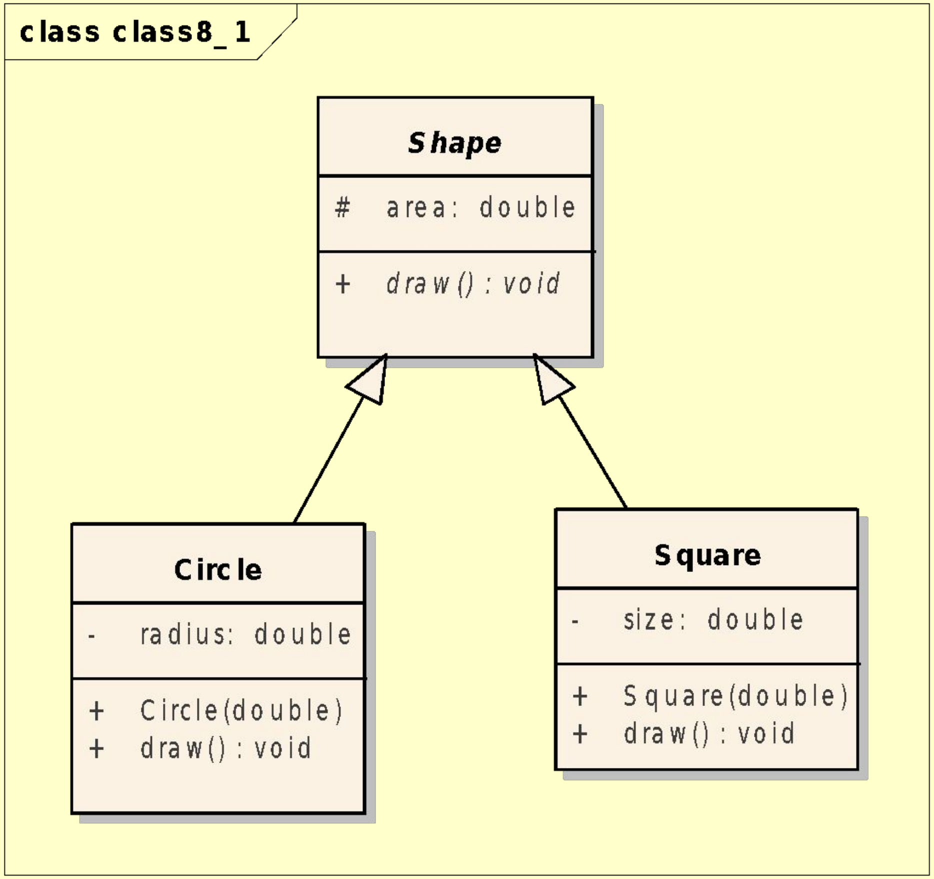
```
public interface MatrixOperations{

   static Matrix add(Matrix a, Matrix b){
      //...
   }

}
```

Helper methods – associated with class, not instances
Cannot be overriden in implementer classes

Abstract Classes

- May contain **abstract** and **implemented methods** as well
- May contain **data**
- **Cannot be instantiated**
- Are designed for subclassing

Abstract Classes



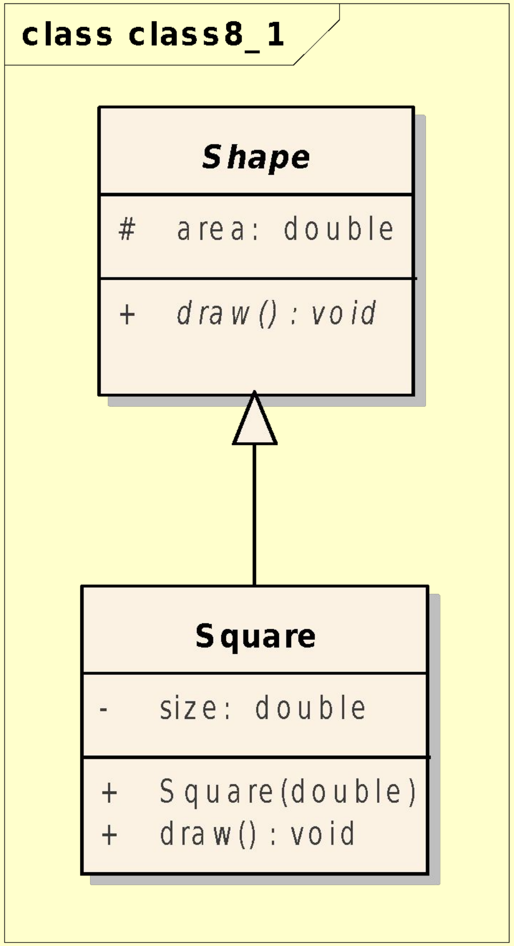
Abstract Classes

```
public abstract class Shape {
    protected double area;
    public abstract void draw();
}
```

```
public class Square extends Shape{
    private double size;

    public Square( double size ){
        this.size = size;
        this.area = size * size;
    }

    @Override
    public void draw() {
        System.out.println("I am a square");
    }

}
```

Abstract Classes vs. Classes

- **Abstract class:**
  - User-defined type
  - Set of data and methods
  - Abstract and implemented methods
  - **Cannot be instantiated**
  - Designed to be subclassed
- **Class:**
  - User-defined type
  - Set of data and methods
  - All the methods are implemented
  - **Can be instantiated**



Abstract Classes vs. Classes vs. Interfaces

	Interface	Abstract class	Class
Abstract method	Yes	Yes	No
Implemented method	No Yes(since Java 8)	Yes	Yes
Attribute (data)	No	Yes	Yes
Constants (final)	Yes	Yes	Yes

Sorting and Interfaces

- Sorting Strings, primitives
  - `Arrays.sort()`
  - `Collections.sort()`
- Sort **user-defined** types
  - The `Comparable` interface
  - The `Comparator` interface

Sorting Collections

- Sorting objects by their **natural order**
    - The **`Comparable`** interface
- Sorting object using a Comparator
    - The **`Comparator`** interface

The `Comparable` interface

```
interface Comparable {
   int compareTo(Object o);
}
```

x.compareTo(y):

- 0: x equal to y
- positive: x > y;
- negative: x< y;

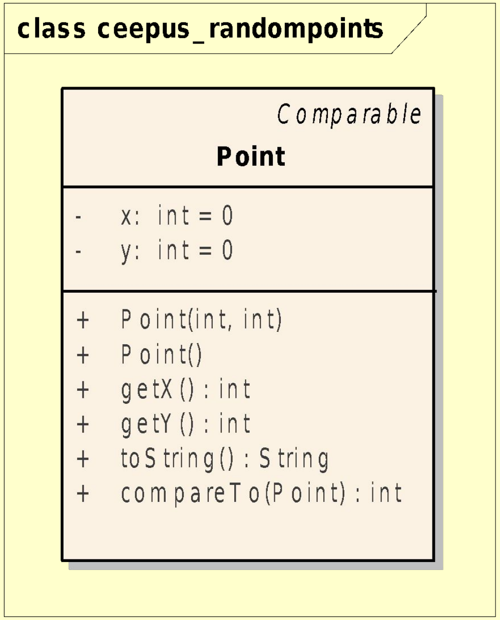
The `Comparable<T>` interface

```
interface Comparable<T> {
   int compareTo(T o);
}
```

Attempts to use a different type are caught at compile time!!!

The Comparable<T> interface

```java
public class Point implements Comparable<Point>{
  //...
  @Override
  public int compareTo(Point o) {
      if( o == null ) throw new NullPointerException();
      if (this.x == o.x && this.y == o.y) {
          return 0;
      }
      if( this.x == o.x) {
          return Integer.compare(this.y, o.y);
      }
      return Integer.compare(this.x, o.x);
  }

}
```



The `Comparable<T>` interface

Consistency

`If a class overrides the equals method, then it is`

advisable (*but not enforced*) that

```
a.equals(b)
```

exactly when

```
a.compareTo(b) == 0
```

The `Comparator<T>` interface

`What if we need multiple sorting criteria?`

- Class **Point**
  - Sorting by `x` then by `y`
  - Sorting by `y` then by `x`
  - Sorting by the distance from the origin `(0,0)`

- For each class we can define **only one natural ordering** through the **Comparable** interface
- We can define an **unlimited number of ordering** using the **Comparator** interface

The `Comparator<T>` interface

```
interface Comparator<T> {
    int compare (T x, T y);
}
```

The `Comparator<T>` interface (1)

```
class DistanceComparator implements Comparator<Point>{
    private final static Point origo = new Point(0,0);

    @Override
    public int compare(Point p1, Point p2) {
        return Double.compare(
                              p1.distanceTo(origo),
                              p2.distanceTo(origo));
    }
}
```

```
ArrayList<Point> points = new ArrayList<Point>();
points.add( new Point(1,2));
points.add( new Point(2,2));
points.add( new Point(1,3));

Collections.sort( points, new DistanceComparator() );
for( Point point: points ){
  System.out.println(point);
}
```

The `Comparator<T>` interface (2)

Anonymous inner class

```
ArrayList<Point> points = new ArrayList<>();
points.add(new Point(1, 2));
points.add(new Point(2, 2));
points.add(new Point(1, 3));
Collections.sort( points, new Comparator<Point>(){
  private final Point origo = new Point(0,0);
  @Override
  public int compare(Point p1, Point p2) {
     return Double.compare(
                    p1.distanceTo(origo),
                    p2.distanceTo(origo));
  }

});
for( Point point: points){
   System.out.println( point );
}
```

The `Comparator<T>` interface (3)

Lambda

```
ArrayList<Point> points = new ArrayList<>();
points.add(new Point(1, 2));
points.add(new Point(2, 2));
points.add(new Point(1, 3));

Collections.sort(points,
    (Point p1, Point p2) ->
        {
            final Point origo = new Point(0,0);
            return Double.compare(p1.distanceTo(origo),
                                  p2.distanceTo(origo));
        }
  );

for (Point point : points) {
    System.out.println(point);
}
```

Module 7
Exceptions

Exceptions

- Define exceptions
- Exception handling: `try`, `catch`, and `finally`
- Throw exceptions: `throw`, `throws`
- Exception categories
- User-defined exceptions

Exception Example

```
public class AddArguments {
   public static void main(String[] args) {
       int sum = 0;
       for( String arg: args ){
           sum += Integer.parseInt( arg );
       }
       System.out.println( "Sum: "+sum );
   }
}
```

```
java AddArguments 1 2 3
Sum: 6
```

```
java AddArguments 1 foo 2 3
Exception in thread "main" java.lang.NumberFormatException: For input string: "foo"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.parseInt(Integer.java:615)
at addarguments.AddArguments.main(AddArguments.java:line_number)
Java Result: 1
```

The `try-catch` statement

```
public class AddArguments2 {
    public static void main(String[] args) {
        try{
            int sum = 0;
            for( String arg: args ){
                sum += Integer.parseInt( arg );
            }
            System.out.println( "Sum: "+sum );
        } catch( NumberFormatException e ){
            System.err.println("Non-numeric argument");
        }
    }
}
```

```
java AddArguments2 1 foo 2 3
Non-numeric argument
```

The `try-catch` statement

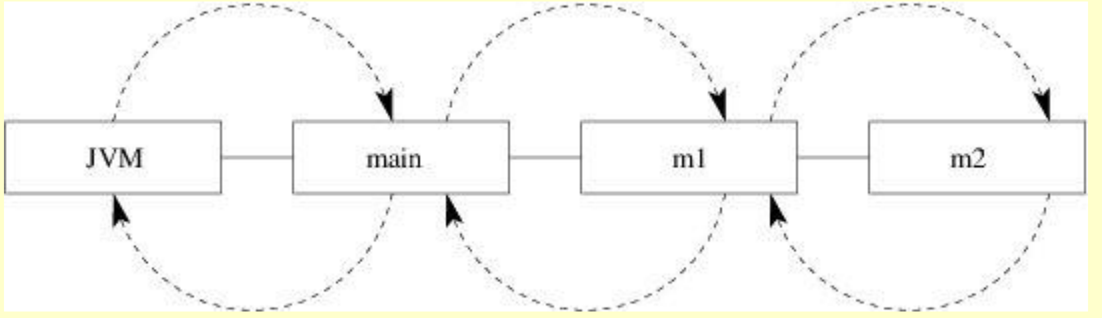
```
public class AddArguments3 {
   public static void main(String[] args) {
         int sum = 0;
         for( String arg: args ){
             try{
              sum += Integer.parseInt( arg );
             } catch( NumberFormatException e ){
                 System.err.println(arg+"is not an integer");
             }
         }
         System.out.println( "Sum: "+sum );
    }
}
```

```
java AddArguments3  1 foo 2 3
foo is not an integer
Sum: 6
```

The `try-catch` statement

```
try{
  // critical code block
  // code that might throw exceptions
} catch( MyException1 e1 ){
  // code to execute if a MyException1 is thrown
} catch( MyException2 e2 ){
  // code to execute if a MyException2 is thrown
} catch ( Exception e3 ){
  // code to execute if any other exception is thrown
} finally{
  // code always executed
}
```

Call Stack Mechanism



- If an exception is not handled in a method, it is thrown to the caller of that method
- If the exception gets back to the main method and is not handled there, the program is terminated abnormally.

Closing resources

The `finally` clause

```
try{
    connectDB();
doTheWork();
} catch( AnyException e ){
    logProblem( e );
} finally {
    disconnectDB();
}
```

The code in the `finally` block is **always executed** (even in case of return statement)

Closing resources

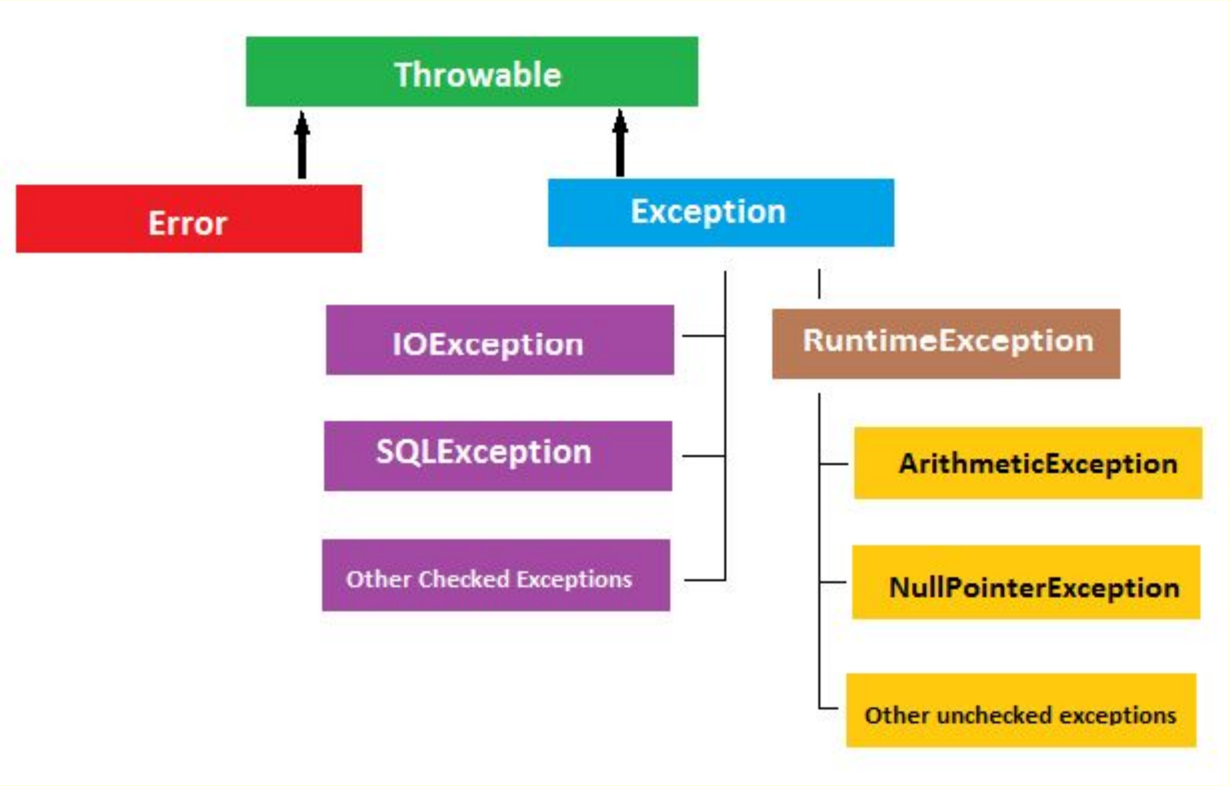
The try-with-resources Statement

- The **try-with-resources** statement **ensures** that each **resource is closed** at the end of the statement.

```
static String readFirstLineFromFile(String path){
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    } catch(Exception e) {
       // exception handling
    }
}
```

Exception Categories

- **Checked** and **unchecked** exceptions



The Handle or Declare Rule

```
public static int countLines( String filename ){
    int counter = 0;
    try (Scanner scanner = new Scanner(new File(filename))){
        while (scanner.hasNextLine()) {
            scanner.nextLine();
             ++counter;
        }
    }catch(FileNotFoundException e){
        e.printStackTrace();
    }
    return counter;
}
```

HANDLE

Usage:

```
System.out.println(ClassName.countLines("input.txt"));
```

The Handle or Declare Rule

```
public static int countLines(String filename) throws
                                     FileNotFoundException {
   try (Scanner scanner = new Scanner(new File(filename))) {
      int counter = 0;
      while (scanner.hasNextLine()) {
         scanner.nextLine();
         ++counter;
      }
      return counter;
   }
}
```

DECLARE
throws

Usage:

```
try{
    System.out.println(ClassName.countLines("input.txt"));
} catch( FileNotFoundException e ){
    e.printStackTrace();
}
```

The `throws` Clause

```
void trouble1 () throws Exception1 {...}

void trouble2 () throws Exception1, Exception2 {...}
```

Principles:

- You do not need to declare runtime (unchecked) exceptions
- You can choose to handle runtime exceptions (e.g. **`IndexArrayOutOfBounds, NullPointerException`**)

Creating Your Own Exceptions

The overriding method can throw:

- No exceptions
- One or more of the exceptions thrown by the overridden method
- One or more subclasses of the exceptions thrown by the overridden method

The overridden method cannot throw:

- Additional exceptions not thrown by the overridden method
- Superclasses of the exceptions thrown by the overridden method

User-Defined Exception

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

User-Defined Exception

```
public class Stack {
    private Object elements[];
    private int capacity;
    private int size;

    public Stack( int capacity ){
        this.capacity = capacity;
        elements = new Object[ capacity ];
    }

    public void push(Object o) throws StackException {
        if (size == capacity) {
            throw new StackException("Stack is full");
        }
        elements[size++] = o;
    }

    public Object top() throws StackException {
        if (size == 0) {
            throw new StackException("Stack is empty");
        }
        return elements[size - 1];
    }
    // ...

}
```

User-Defined Exception

```
Stack s = new Stack(3);
for (int i = 0; i < 10; ++i) {
   try {
       s.push(i);
    } catch (StackException ex) {
           ex.printStackTrace();
    }
}
```

Best practices to handle exceptions

- Clean up resources in a **finally** block or use a **try-with-resource** statement
- Prefer specific exceptions
- Don't ignore exceptions
- Don't log and throw. Instead, wrap the exception without consuming it
- Catch early, handle late

Source: [9 Best Practices to Handle Exceptions in Java]

Module 8

Nested Classes

Nested Classes

- **When?**
  - If a class is used only inside of another class (encapsulation)
  - Helper classes

Nested Classes

- **The place of nesting**
  - Class
  - Method
  - Instruction
- **Embedding method**
  - Static
  - Non-static

Static Nested Class

```
public class Slist{
  private Element head;

  public void insertFirst( Object value ){
     head = new Element(value, head);
  }

  private static class Element{
     private Object value;
     private Element next;
     public Element( Object value, Element next){
        this.value = value;
        this.next = next;
     }
     public Element( Object value){
        this.value = value;
        this.next = null;
     }
  }
}
```

Used only inside the **Slist** class

The Iterator interface

Package: `java.util`

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

Make Slist iterable using the Iterator interface!!!

The Iterator interface

```
Slist list = new Slist();
for( int i=0; i<10; ++i ){
    list.insertFirst( i );
}

Iterator it = list.createIterator();
while( it.hasNext() ){
   System.out.println( it.next() );
}
```

Factory Method
Design Pattern

1. Solution – Non-static Nested Class

```
public class Slist{
  private Element head;
  //...

  public Iterator createIterator(){
     return new ListIterator();
  }

  private class ListIterator implements Iterator{
     private Element act = head;
     public boolean hasNext(){
         return act != null;
     }
     public Object next(){
        Object value = act.value;
        act = act.next;
        return value;
     }
  }
}
```

Relation between Slist and ListIterator objects

1. Solution – Non-static Nested Class

```
public class Slist{
  private Element head;
  //...

  public Iterator createIterator(){
     return new ListIterator();
  }

  private class ListIterator implements Iterator{
     private Element act = head;
     public boolean hasNext(){
         return act != null;
     }
     public Object next(){
        Object value = act.value;
        act = act.next;
        return value;
     }
  }
}
```

Class ListIterator is used only once!!!

2. Solution – Anonymous Inner Class

```
public class Slist{
  private Element head;
  //...

  public Iterator createIterator(){
     return new Iterator(){
        private Element act = head;

        public boolean hasNext(){
           return act != null;
        }

        public Object next(){
           Object value = act.value;
           act = act.next;
           return value;
        }
     };
}
```

Module 9
Threads

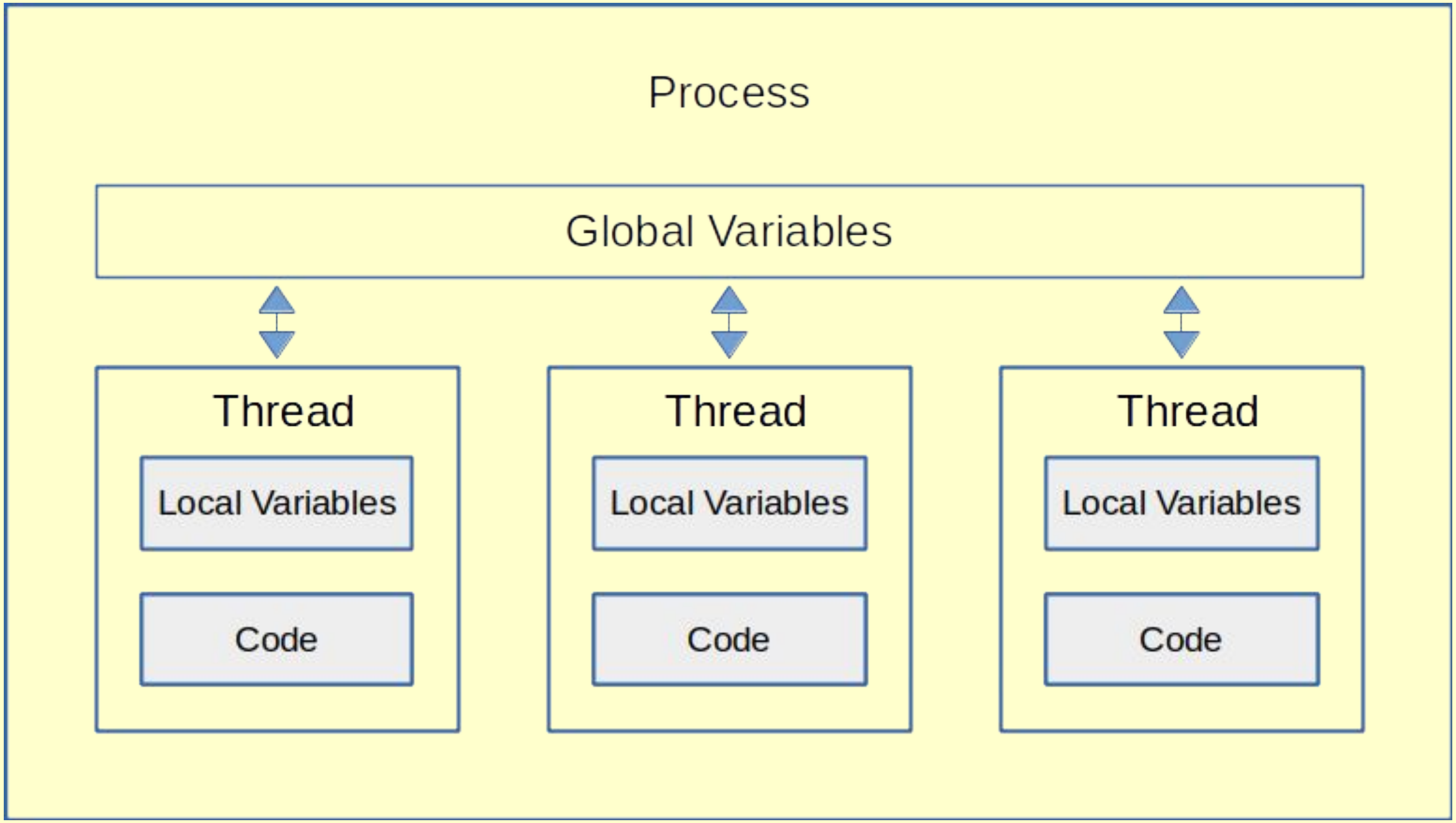
Outline

- Definition
- Creation: Thread and Runnable
- Synchronization
- Executors and thread pools

What are threads?

- **Operating Systems**
  - lightweight process
  - runs in the address space of a process
  - has its own program counter (PC)+stack
  - shares code and data with other threads
- **Object-oriented Programming**
  - an object – an instance of the class Thread

What are threads?



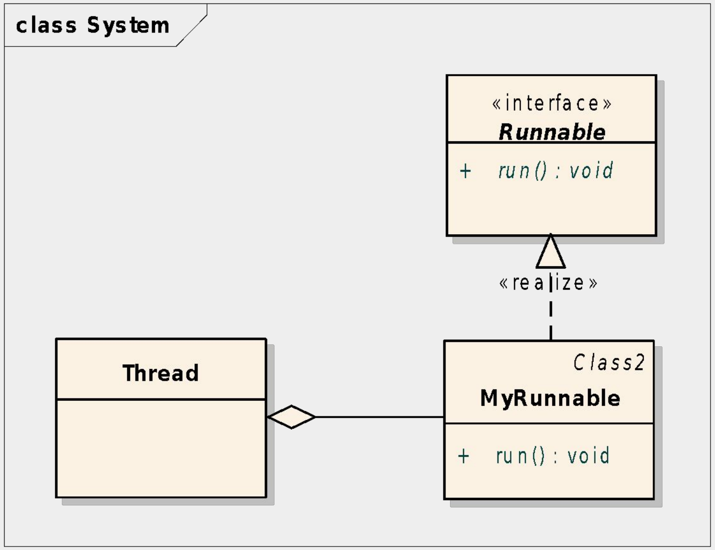
Threads

`java.lang.Thread` = Infrastructure(PC+Stack)

`java.lang.Runnable` = Code

Thread's creation (1)

```
public class  MyRunnable implements Runnable{
  private int id;

  public MyRunnable(int id ){
    this.id = id;
  }

  public void run(){
    for( int i=0; i<10; ++i){
      System.out.println(“Hello”+id+” “+i);
    }
  }
}

…
MyRunnable r = new MyRunnable(1);
Thread t = new Thread( r );
```



Starting the thread

```
Thread t = new Thread( r );
```

Constructor initializes the thread object

```
t.start();
```

Calls the thread object's run method

Thread's creation (1)

```
public class Test{
  public static void main(String args[]){
      Thread t1 = new Thread( new MyRunnable(1));
      Thread t2 = new Thread( new MyRunnable(2));
      t1.start();
      t2.start();
  }
}
```

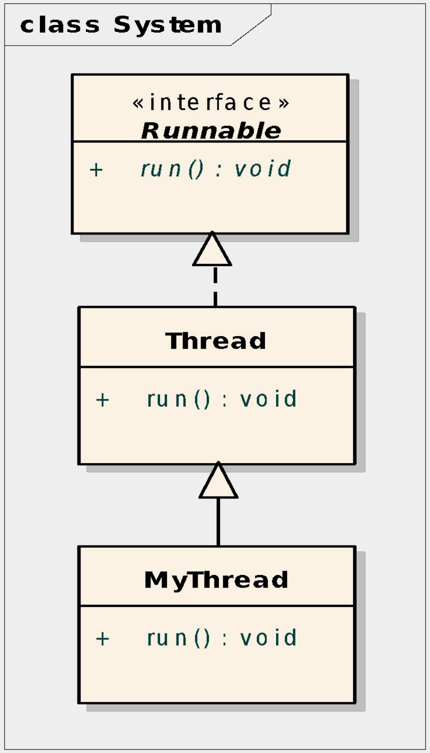
Output?

Thread's creation (2)

```
class MyThread extends Thread {
    private int id;

    public MyThread(int id) {
        this.id = id;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; ++i) {
            System.out.println("Hello" + id + " " + i);
        }
    }
}

…
Thread t = new MyThread(1);
t.start();
```

Thread's creation (2)

```
public class Test {
    public static void main(String[] args) {
        Thread t1 = new MyThread(1);
        Thread t2 = new MyThread(2);
        t1.start();
        t2.start();
    }
}
```



Example (1)

```
public class MyFirstRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("In a thread");
    }
}

Usage:

Thread thread = new Thread(new MyFirstRunnable());
thread.start();
System.out.println("In the main Thread");
```

Output?

Example (2)

```
public class MyFirstRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("In a thread");
    }
}
```

Usage:

```
Runnable runnable = new MyFirstRunnable();
for(int i = 0; i<25; i++){
    new Thread(runnable).start();
}
```

How many threads?

Example (3)

```
public class MyFirstRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("In a thread");
    }
}
Usage:

Thread thread = new Thread(new MyFirstRunnable());
thread.run();
System.out.println("In the main Thread");
```

Output?

Operations on threads

- make the current Thread **sleep**
- wait for another thread to complete (**join**)
- manage the **priorities** of threads
- **interrupt** a thread

sleep()

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e){
    e.printStackTrace();
}
```

sleep()

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e){
    e.printStackTrace();
}
```

- It always pause the current thread execution.
- The actual time thread sleeps depends on system timers and schedulers (for a busy system, the **actual time** for sleep is a little bit **more than** the specified **sleep time**).

join()

```
Thread t2 = new Thread(new R());
t2.start();
try {
    t2.join();
} catch (InterruptedException e){
    e.printStackTrace();
}
```

setPriority()/getPriority()

```
public class ThreadPriorityRange {
    public static void main(String[] args) {
        System.out.println("Minimal priority : " +
                                Thread.MIN_PRIORITY);
        System.out.println("Maximal priority : " +
                                Thread.MAX_PRIORITY);
        System.out.println("Norm priority : " +
                                Thread.NORM_PRIORITY);
         }
}
```

interrupt()

A thread can be interrupted:

- **if the thread is sleeping**
- **if the thread is waiting for another thread to join**

interrupt()

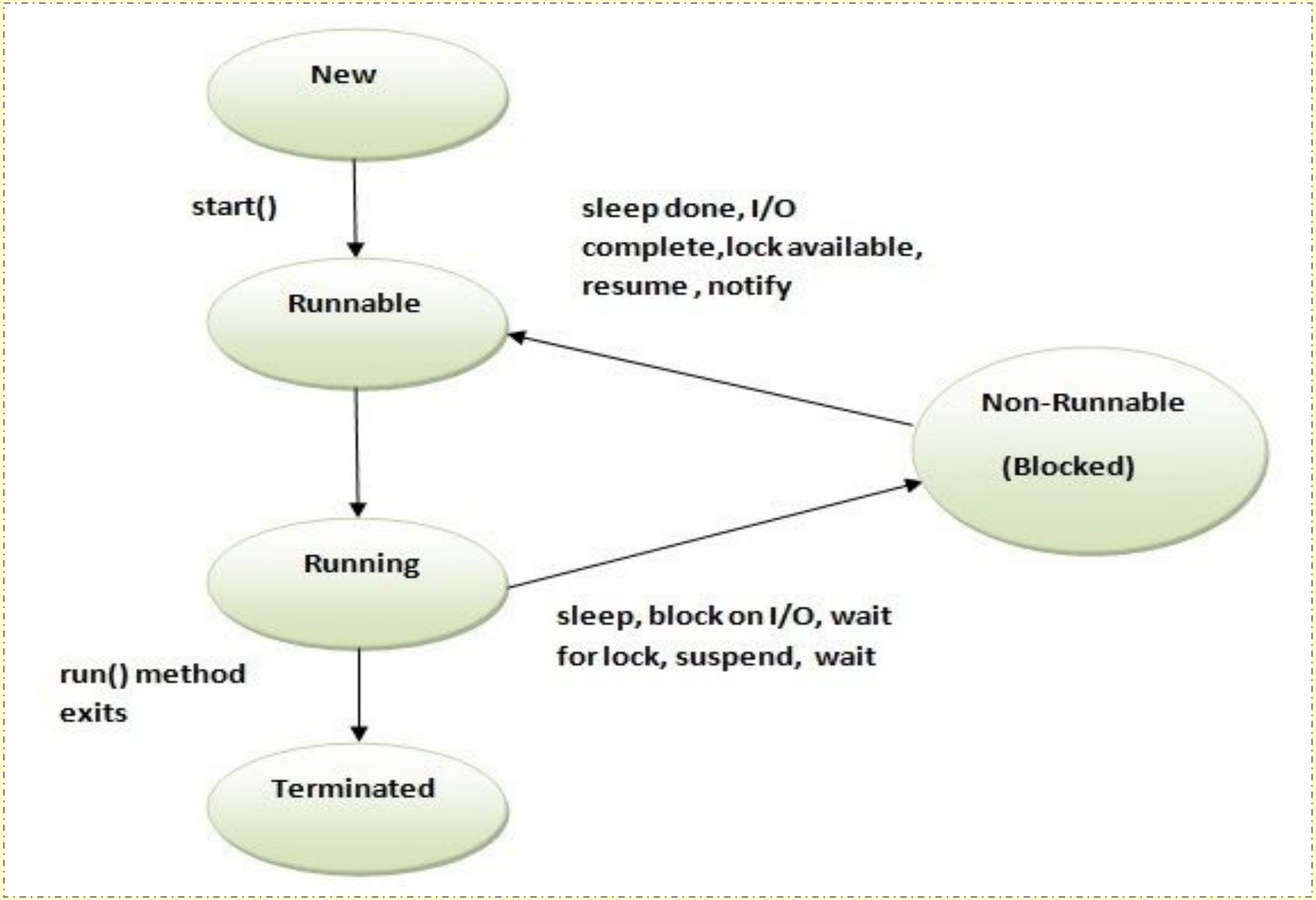
```
private static class ForeverRunnable implements Runnable {
  public void run() {
    while (true) {
       System.out.println(Thread.currentThread().getName() +
                        ": " + System.currentTimeMillis());
       try {
         Thread.sleep(5000);
       } catch (InterruptedException e) {
         System.out.println(
            Thread.currentThread().getName() +
                                 "has been interrupted");
       }
    }
  }
}
```

interrupt()

```
private static class ForeverRunnable implements Runnable {
  public void run() {
    while (true) {
       System.out.println(Thread.currentThread().getName() +
                        ": " + System.currentTimeMillis());
       try {
             Thread.sleep(5000);
       } catch (InterruptedException e) {
             System.out.println( Thread.currentThread().getName() +
                                 "has been interrupted");
       }
    }
  }
}
```

```
public static void main(String[] args) {
  Thread t2 = new Thread(new ForeverRunnable());
  System.out.println("Current time millis : " +
                             System.currentTimeMillis());
  t2.start();
  t2.interrupt();
}
```

Thread's states



Need for synchronization

Thread1

```
public class Counter {
    private int value = 0;

    public int getNextValue(){
        return value++;
    }
}
```

Need for synchronization

Thread1

```
public class Counter {
    private int value = 0;

    public int getNextValue(){
        return value++;
    }
}
```

Thread2

Need for synchronization

```
class Counter {
    private int value;

    public int getNextValue() {
        return ++value;
    }
    public int getValue(){
        return value;
    }
}
```

Need for synchronization

```
Runnable task = new Runnable() {
  @Override
  public void run() {
      for( int i=0; i<10000; ++i) {
          counter.getNextValue();
      }
  }
};
```

Need for synchronization

```
Counter counter = new Counter();
Thread t1 = new Thread(task);
Thread t2 = new Thread(task);
t1.start();
t2.start();
try{
   t1.join();
   t2.join();
} catch( InterruptedException e ){
}
System.out.println("COUNTER: "
                        +counter.getValue());
```

Output?

Need for synchronization

`value++` <--- Not atomic!

1. Read the current value of "value"
2. Add one to the current value
3. Write that new value to "value"

Solution (1)

```
public class Counter {
    private int value = 0;

    public synchronized int getNextValue(){
            return value++;
    }
}
```

Solution (2)

```
public class Counter {
        private int value = 0;

        public int getNextValue(){
                synchronized(this){
                    value++;
                }
                return value;
        }
}
```

Solution (3)

```
import java.util.concurrent.atomic.AtomicInteger;

public class Counter {
   private AtomicInteger value = new AtomicInteger(0);

   public int getNextValue(){
       return value.incrementAndGet();
   }

   public int getValue(){
       return value.intValue();
   }

}
```

Synchronized Blocks

- every object contains a single **lock**
- the **lock** is taken when synchronized section is entered
- if the **lock** is not available, thread enters a waiting queue
- if the **lock** is returned, thread is resumed

Thread Safe

- A class is **thread safe** if it behaves always in the same manner when accessed from multiple threads.
- Stateless objects (**immutable classes**) are always thread safe:
  - String
  - Long
  - Double

Executors and thread pools

CPU cores

Linux:

CPU info

```
$cat /proc/cpuinfo
```

CPU cores info

```
$top
```

then press **1**

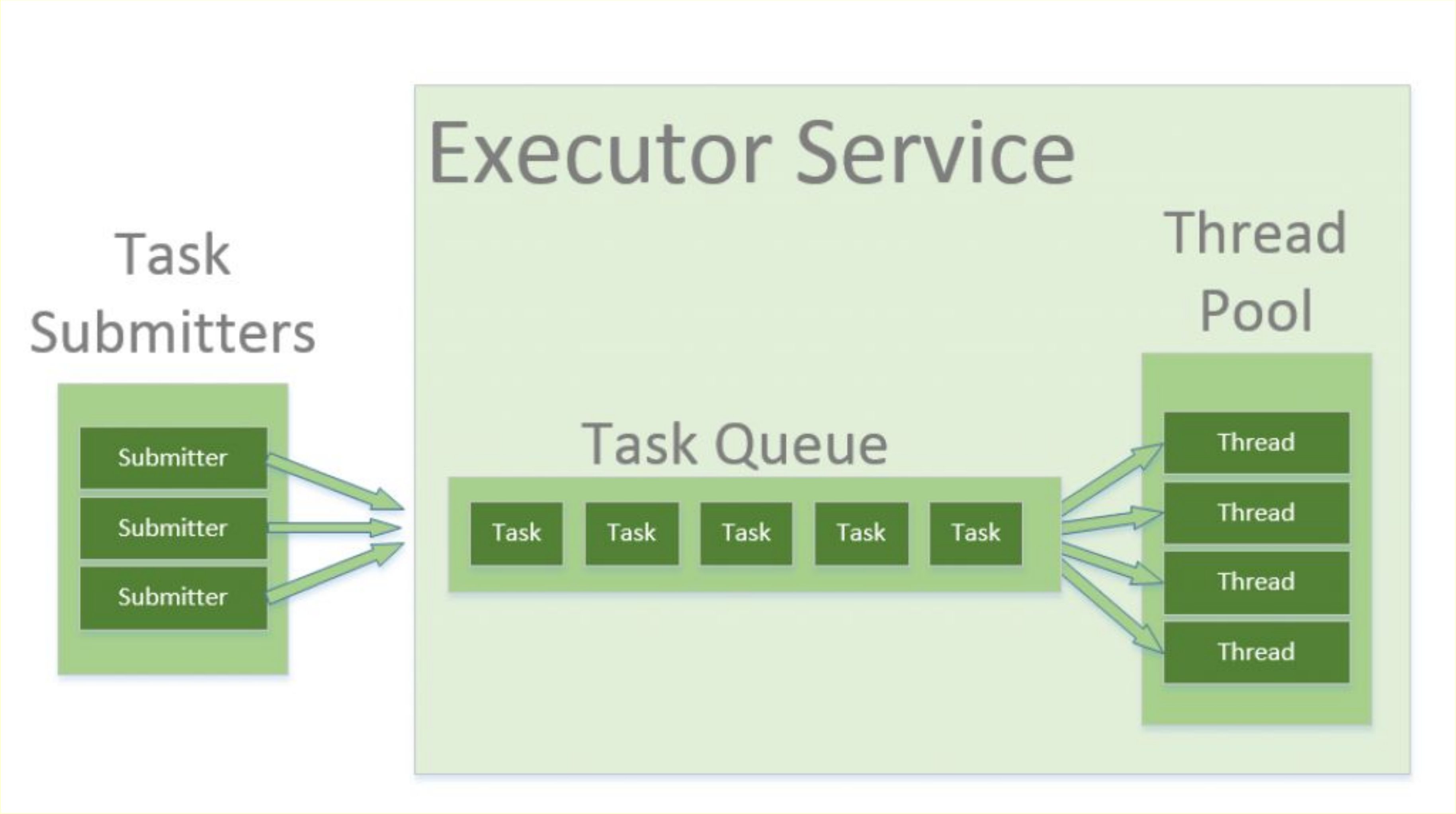
Thread pool

- In **Java threads** are mapped to **system-level threads** (Operating system resources)

- When you use a thread pool, write your concurrent code in the form of parallel tasks and submit them for execution to an instance of a thread pool.

https://www.baeldung.com/thread-pool-java-and-guava

Thread Pool



Executor Service
Task Submitters
Submitter
Submitter
Submitter
Task Queue
Task
Task
Task
Task
Task
Thread Pool
Thread
Thread
Thread
Thread

ExecutorService

```
// number of increments
int n = 10000;
Counter counter = new Counter();

ExecutorService executor = Executors.newFixedThreadPool(2);
Runnable task = new Runnable() {
   @Override
   public void run() {
       for( int i=0; i<n; ++i) {
           counter.getNextValue();
       }
   }
};
executor.execute( task );
executor.execute( task );
executor.shutdown();
try {
   executor.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Counter: " + counter.getValue());
```

Module 10
GUI Programming
Swing and JavaFx

Java GUIs

- **AWT (Abstract Windowing Toolkit)** – since JDK 1.0
  - Uses native control
  - Appearance/behavior depends on platform
- **Swing** – since JDK 1.2
  - Implemented completely in Java (light weight)
- **JavaFX** – since JDK 8
  - Written as a native library
  - Provided on a wide variety of devices
- **SWT (Standard Widget Toolkit)**
  - Eclipse

GUI Programming

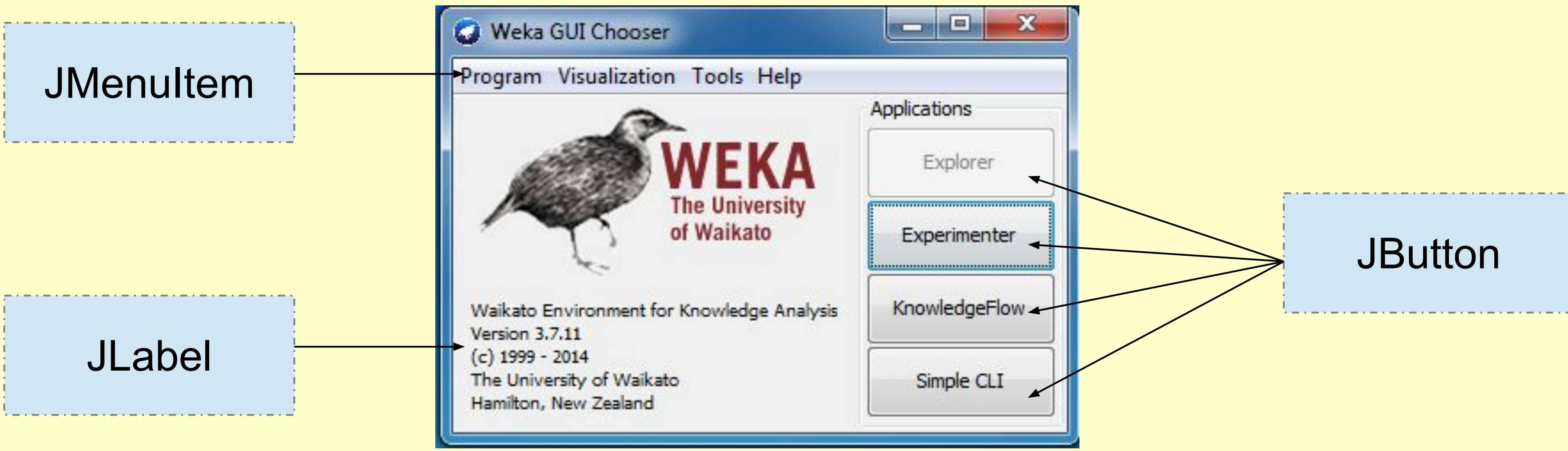
Swing

Outline

- ***Containers, components*** **and** ***layout managers***
- `FlowLayout`, `BorderLayout`, and `GridLayout`
- Add components to a container
- Events and event handling
- Delegation model
- Adapter classes

Component

- Represents an object with *visual* representation
- Other names for components: widgets, controls

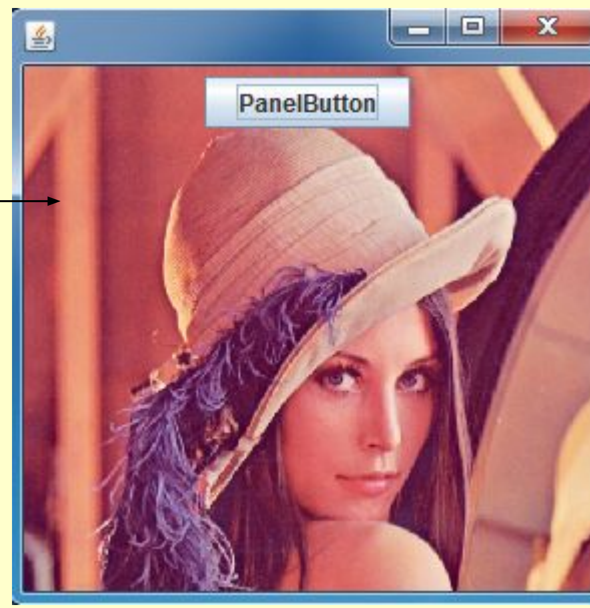


Container

- A special component that holds other components
- Used for grouping other components

JPanel

contains a picture



JFrame

The first GUI program

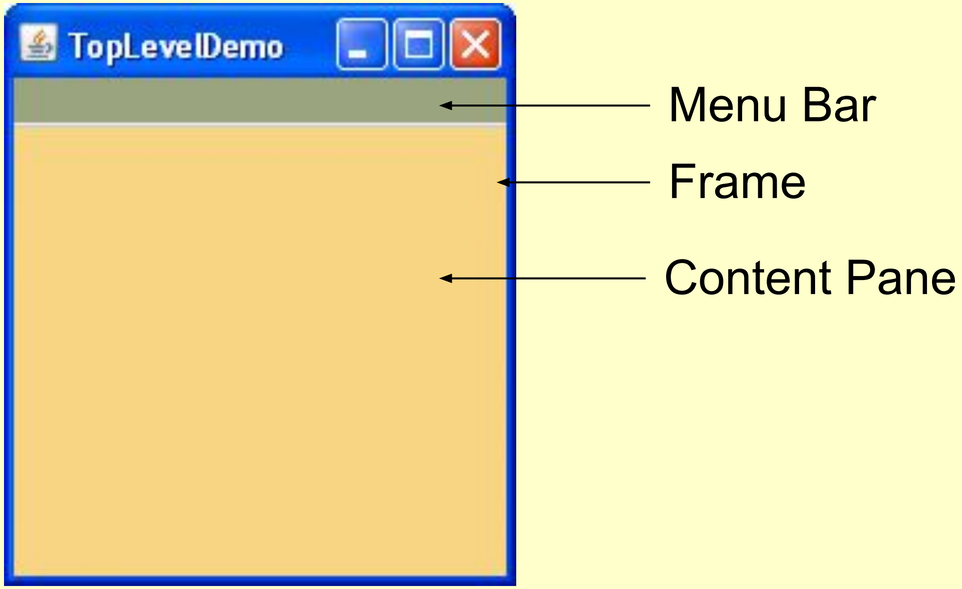
```
public static void main(String[] args) {
        JFrame f = new JFrame("The First Swing
                                        Application");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setBounds( 100,100, 300, 300);
        f.setVisible(true);
  }
```

Frames

JFrame

- Top level container
  - can have menu bars
- Contains a JRootPane
- Have title and resizing corners
- Have BorderLayout as

  the default layout manager



Positioning Components

- Responsibility of the layout manager
  - **size** (dimension: width and height in pixels)
  - **position** (location of the top left corner)
- You can disable the layout manager:

  setLayout(null),

  then use

  - `setSize() + setLocation()`
  - `setBounds()`

Organizing Components (1)

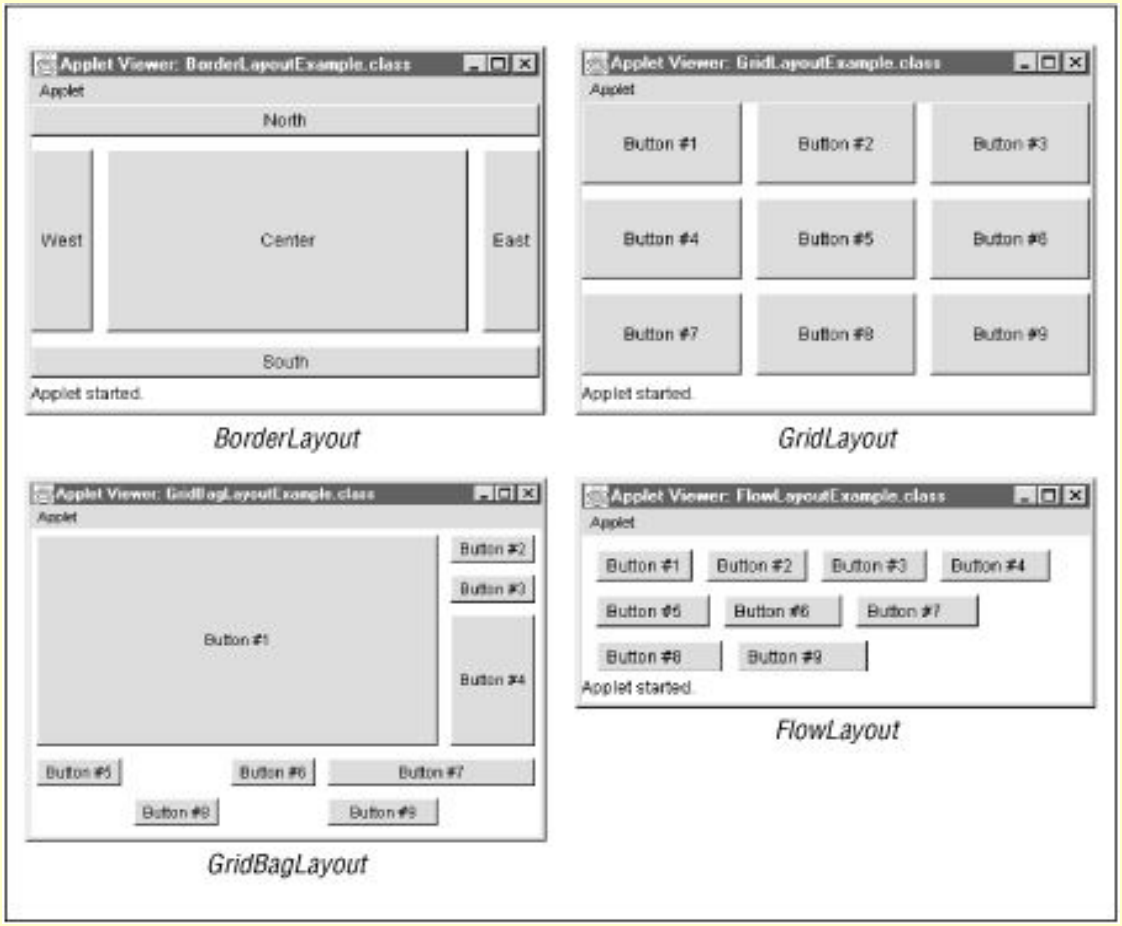
```
JFrame f = new JFrame("The First Swing Application");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel p = new JPanel();
p.setBackground(Color.blue);
JButton b = new JButton("Yes");
p.add(b);
f.setContentPane(p);

f.setBounds( 100,100, 300, 300);
f.setVisible(true);
```

Organizing Components (2)

```
JFrame f = new JFrame("The First Swing Application");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel p = new JPanel();
p.setBackground(Color.blue);
p.setLayout( null );
JButton b = new Jbutton("Yes");
b.setSize(100,60);
b.setLocation(200, 200);
p.add(b);
f.setContentPane(p);

f.setBounds( 100,100, 300, 300);
f.setVisible(true);
```

Layout Managers

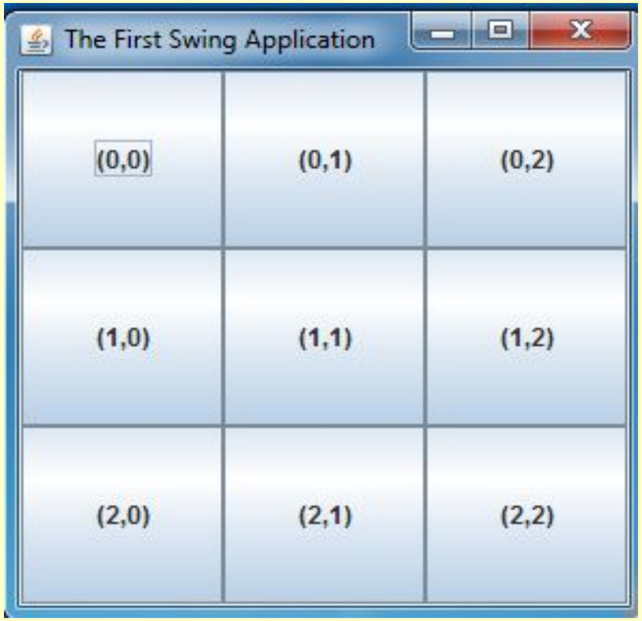
- FlowLayout
- BorderLayout
- GridLayout
- GridBagLayout



Layout Managers

GridLayout

```
public static JPanel createPanel( int n){
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout( n, n));
    for( int i=0; i<n; ++i){
        for( int j=0; j<n; ++j){
            panel.add( new JButton
                  ("("+i+","+j+")"));
        }
    }
    return panel;
}
```



Creating UI

- Aggregation:
  - Frame aggregation
- Inheritance:
  - Frame inheritance

Creating UI

Aggregation

```
public class FrameAggregation {

    private static void initFrame() {
      JFrame frame = new JFrame("FrameAggregation");
      frame.add(new JButton("Ok"), "Center");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setBounds(100, 100, 200, 200);
      frame.setVisible(true);
    }

    public static void main(String[] args) {
        initFrame();
    }
}
```

Creating UI

Inheritance

```
public class FrameInheritance extends JFrame {
    private JButton button;
    public FrameInheritance(){
        initComponents();
    }
    private void initComponents() {
      this.setTitle("FrameInheritance");
        this.add(new JButton("Ok"), "Center");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setBounds(100, 100, 200, 200);
        this.setVisible(true);
    }
    public static void main(String[] args) {
       new FrameInheritance();
    }
}
```

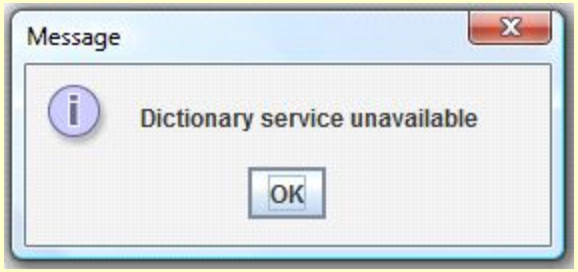
Menus

```
private static JMenuBar createMenu() {
   //Menubar
   MenuBar menuBar; JMenu filemenu, helpmenu;
   JMenuItem  menuItem;
   menuBar = new JMenuBar();
  // Build File menu.
  filemenu = new JMenu("File"); menuBar.add(filemenu);
  menuItem = new JMenuItem("New"); filemenu.add(menuItem);
  menuItem = new JMenuItem("Exit"); filemenu.add(menuItem);
  // Build Help menu.
  helpmenu = new JMenu("Help");
  menuBar.add(helpmenu);
  menuItem = new JMenuItem("About");
  helpmenu.add(menuItem);
  return menuBar;
}

frame.setJMenuBar(createMenu());
```

Dialogs

JOptionPane (1)

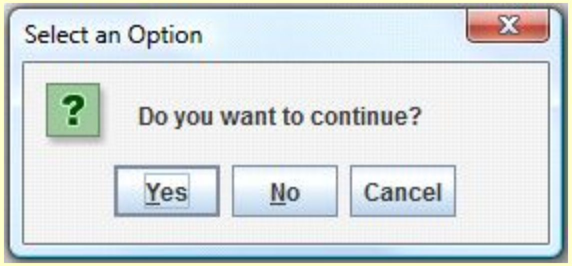
```
JOptionPane.showMessageDialog(
    Component parent, String message);
```



Dialogs

JOptionPane (2)

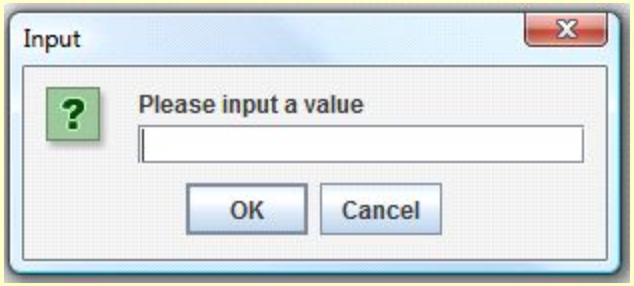
```
int result =
JOptionPane.showConfirmDialog(
            Component parent, String message);
Result:
    YES_OPTION (0), NO_OPTION (1), CANCEL_OPTION (2)
```



Dialogs

JOptionPane (3)

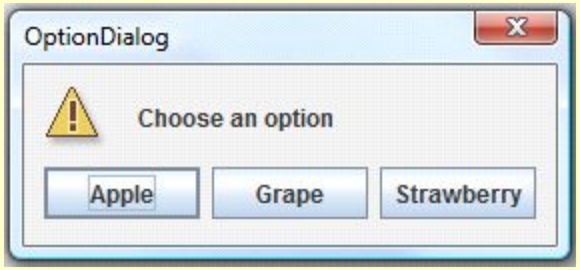
```
String value=
    JOptionPane.showInputDialog("Please input a value");
```



Dialogs

JOptionPane (4)

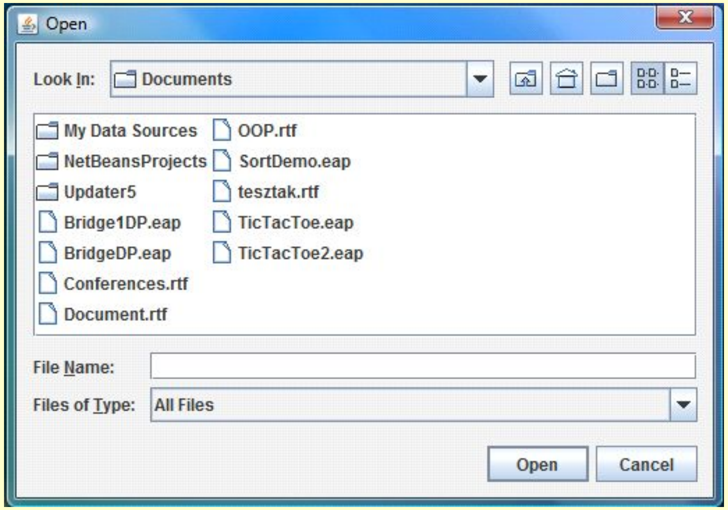
```
String options[]={"Apple", "Grape", "Strawberry"};

int res = JOptionPane.showOptionDialog(form, "Choose an
option", "OptionDialog",JOptionPane.DEFAULT_OPTION,
JOptionPane.WARNING_MESSAGE,null, options, options[0]);
```



Dialogs

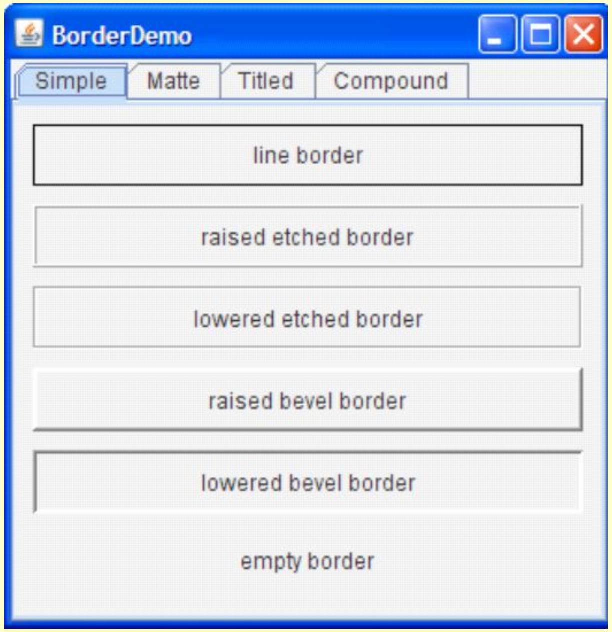
Chooser

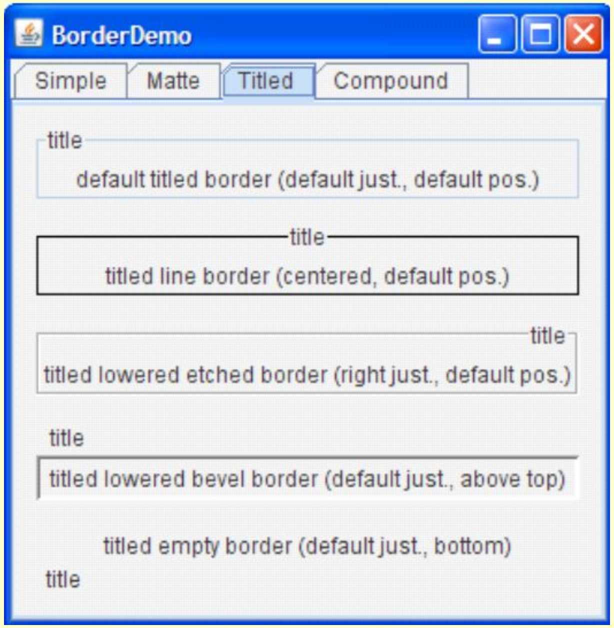
```
JFileChooser chooser = new JFileChooser();
int returnVal = chooser.showOpenDialog(parent);
if(returnVal == JFileChooser.APPROVE_OPTION) {
      System.out.println(
   "You chose to open this file: " +
         chooser.getSelectedFile().getName());
}
```



Borders

```
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createLineBorder(Color.black));
```





http://docs.oracle.com/javase/tutorial/uiswing/components/border.html

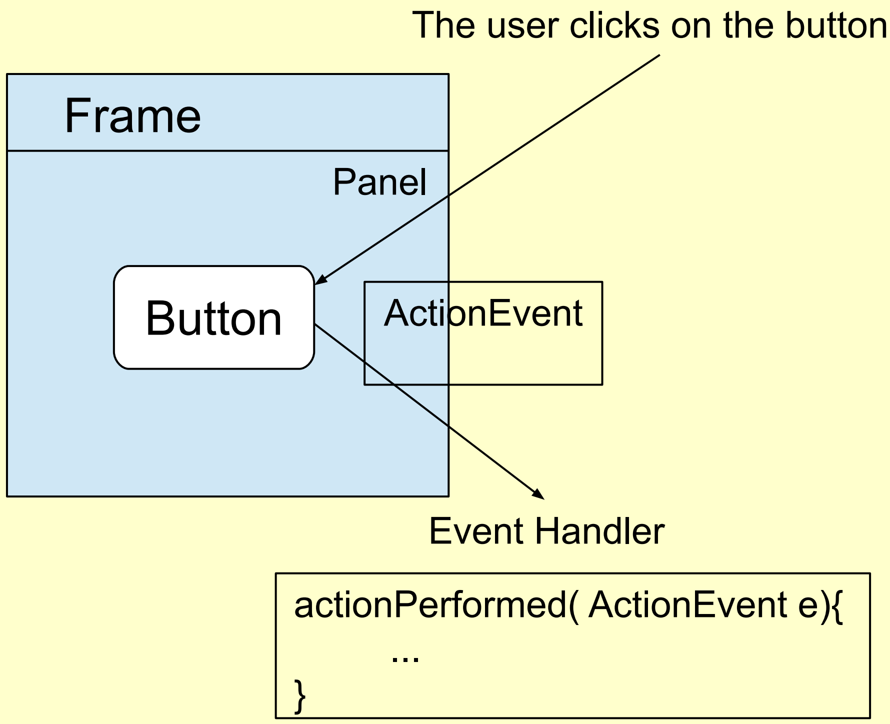
Custom properties

- (key, value) pairs associated to **`JComponent`** type objects
  - Key: **`Object`**
  - Value: **`Object`**

```
JButton button = new JButton("Press Me");
button.putClientProperty("order","10");
//...
button.getClientProperty("order");
```

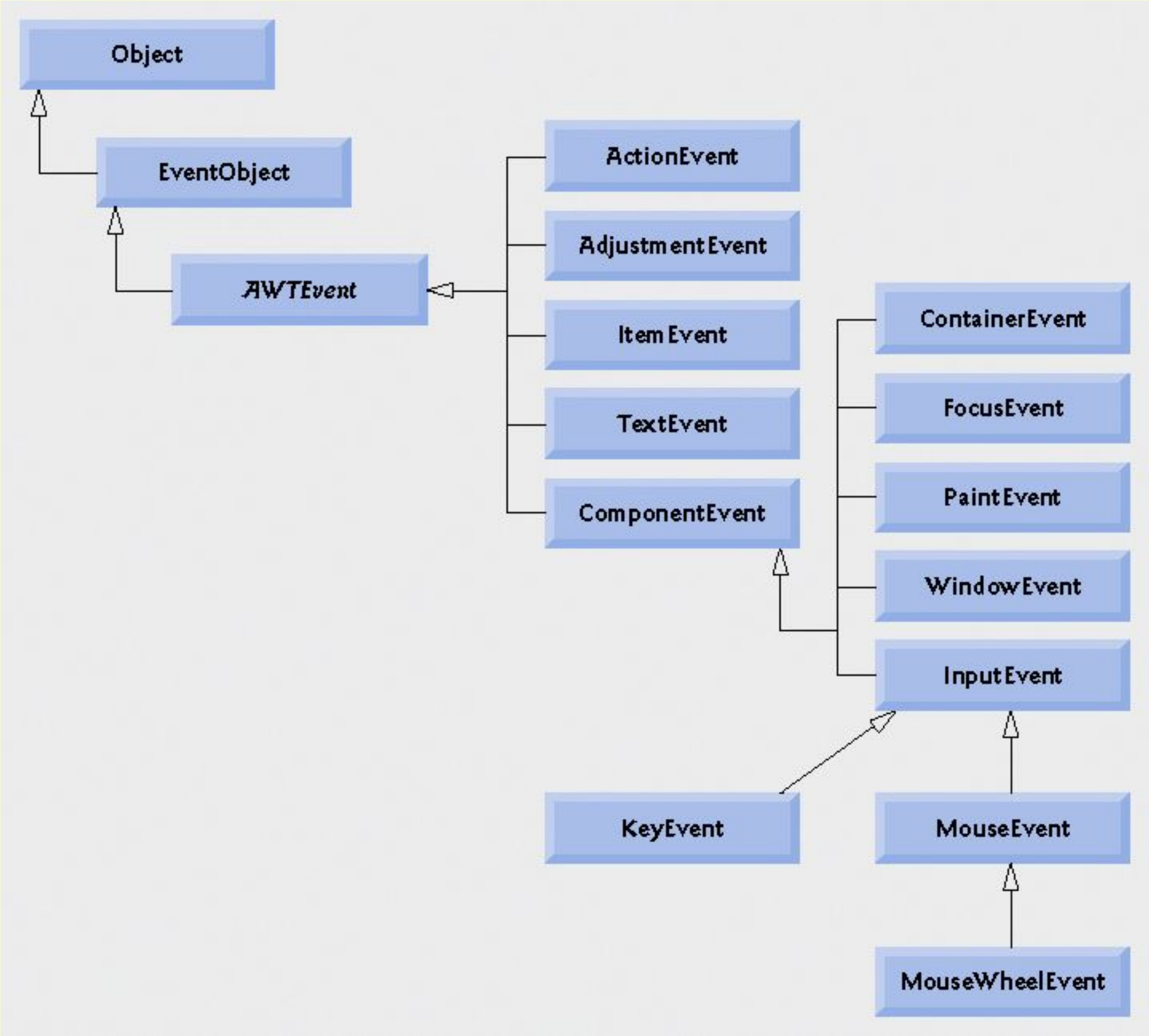
Event Handling

- **Event** – objects that describe what happened
- **Event source** – the generator of an event
- **Event handler** – a method that
  - receives an event object,
  - deciphers it,
  - and processes the user's interaction



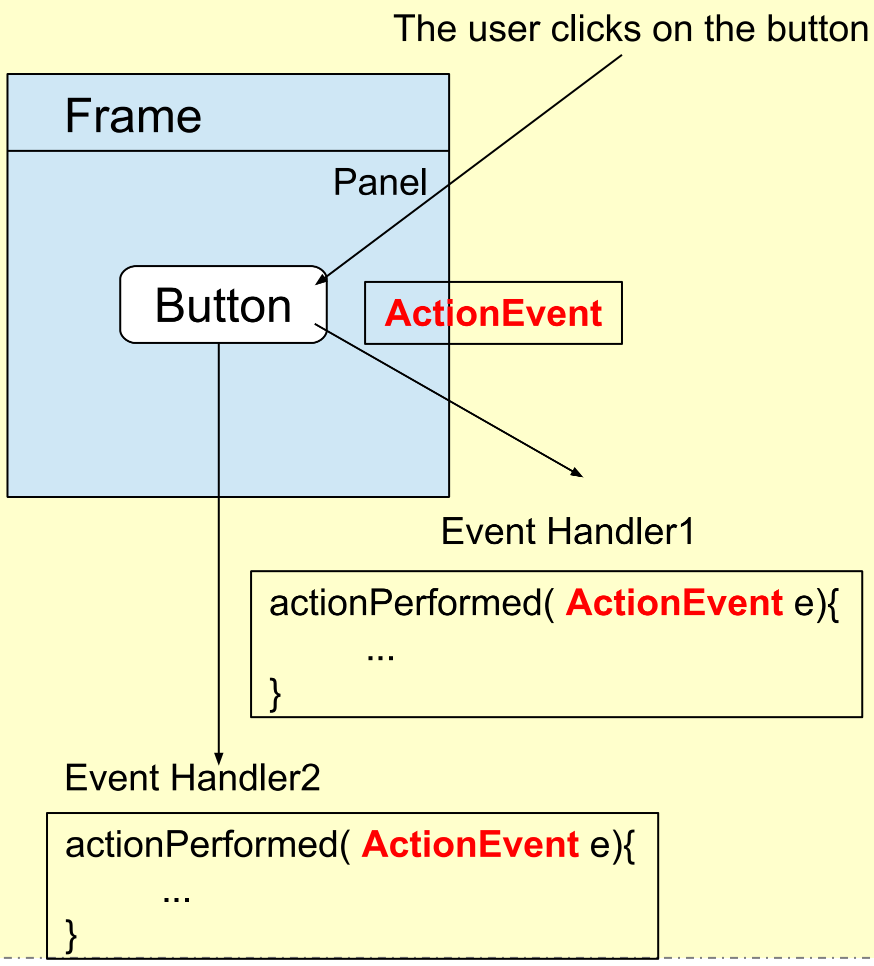
Event Types

- Low level
  - Window
  - Keyboard
  - Mouse
- High level
  - ActionEvent
  - ItemEvent



Event Handling

- *One event* – many *handlers*
- Event handlers are registered by event source components



Delegation Model

- Client objects (handlers) register with a GUI component that they want to observe
- GUI components trigger the handlers for the type of event that has occurred
- Components can trigger more than one type of events

Delegation Model

Event handler

```
JButton b = new JButton("Yes");
f.add( b );
b.addActionListener( new ActionListener(){
    @Override
    public void actionPerformed(ActionEvent e) {
        if( b.getText().equals("Yes")){
            b.setText("No");
        }else{
            b.setText("Yes");
        }
    }
});
```

Event source

(I) Definition of an **anonymous inner class** which implements ActionListener interface
(II) Creation of an instance from that anonymous inner class
(III) This instance is responsible for event handling

Delegation Model

Java 8 - Lambdas

```
JButton b = new JButton("Yes");
f.add( b );
b.addActionListener(e->
 {
   b.setText( b.getText().equals("No") ? "Yes": "No");
 }
);
```

ActionEvent

Many sources – One listener

```
public class MyFrame implements ActionListener{
   // ...
    public void initComponents(){
      for( int i=0; i<n; ++i){
        for( int j=0; j<n; ++j){
                JButton b = new JButton("");
            panel.add( b);
            b.addActionListener( this );
        }
      }
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        JButton source = (JButton) e.getSource();
        source.setBackground(Color.red);
   }
}
```

Example

Custom Component

```
public class DrawComponent extends JComponent{
   private ArrayList<Point> points= new ArrayList<Point>();
   private Color color = Color.red;

   public DrawComponent(){
     this.addMouseListener(new MouseAdapter(){
        @Override
        public void mousePressed(MouseEvent e) {
           points.clear();
           points.add( new Point( e.getX(), e.getY()));
        }
     });
     this.addMouseMotionListener(new MouseMotionAdapter(){
        @Override
        public void mouseDragged(MouseEvent e) {
           points.add( new Point( e.getX(), e.getY()));
           DrawComponent.this.repaint();
        }
     });
   }
   …
}
```

Example

Custom Component

```
public class DrawComponent extends JComponent{
    //...
    @Override
    public void paint(Graphics g) {
        g.setColor(color);
        if( points != null && points.size()>0){
            Point startPoint = points.get(0);
            for( int i=1; i<points.size(); ++i ){
             Point endPoint = points.get(i);
                g.drawLine(startPoint.x, startPoint.y,
                           endPoint.x, endPoint.y);
                startPoint = endPoint;
            }
        }
    }

    public void clear(){
        points.clear();
        repaint();
    }
}
```

Event listeners

- **General listeners**
  - `ComponentListener`
  - `FocusListener`
  - `MouseListener`
- **Special listeners**
  - `WindowListener`
  - `ActionListener`
  - `ItemListener`

Event adapter classes

- **Problem:**
  - Sometimes you need only one event handler method, but the listener interface contains several ones
  - You have to implement all methods, most of them with empty ones
- **Solution:**
  - An Event Adapter is a convenience class
  - Implements all methods of a listener interface with empty methods
  - You extend the adapter class and override that specific method

Event Adapter Classes

Example

```
public class MyClass extends JFrame {
    ...
    someObject.addMouseListener(
        new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
           //Event listener implementation
        }
    });

}
```

GUI Programming
JavaFX

Sources:

- https://docs.oracle.com/javafx/2/events/jfxpub-events.htm
- http://tutorials.jenkov.com/javafx
- https://www.tutorialspoint.com/javafx

Outline

- Creating UI
  - Declarative UI - FXML
  - Programmatic - Java
- Event Handling

Cross-platform

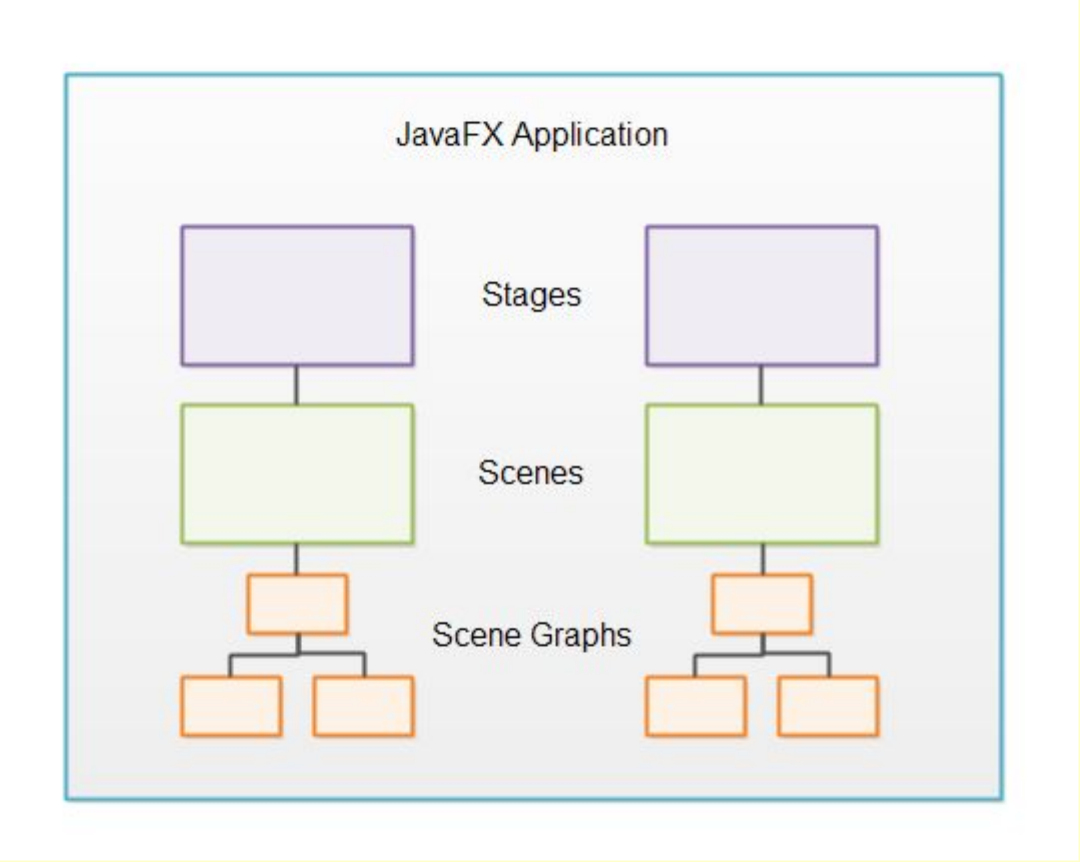
JavaFX can run on:

- Windows
- Linux
- Mac
- iOS
- Android

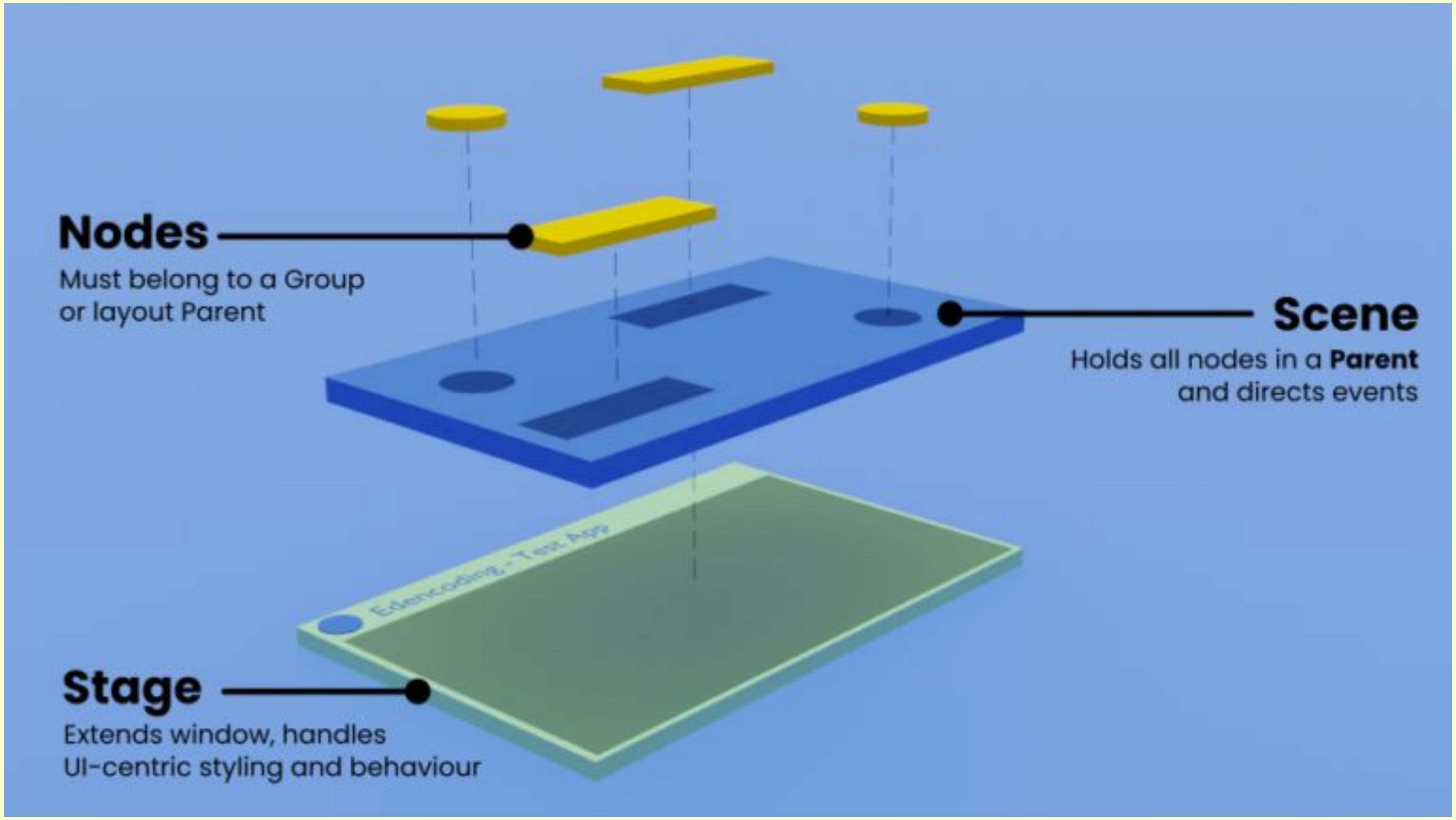
GUI components

Core	**Layout Panes (Containers)**	**Basic Controls**
Stage, Scene, Node, Properties, FXML	HBox, VBox, BorderPane, StackPane GridPane, FlowPane, TilePane, ...	Label, Button, TextField, ListView, DatePicker, FileChooser, ...
Web	**Other concepts**	
WebView, WebEngine	Font, Canvas, Animation, Video, ...	

JavaFX overview

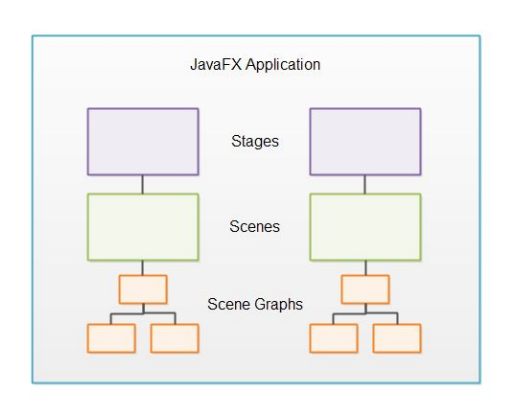


Stage vs Scene



[source]

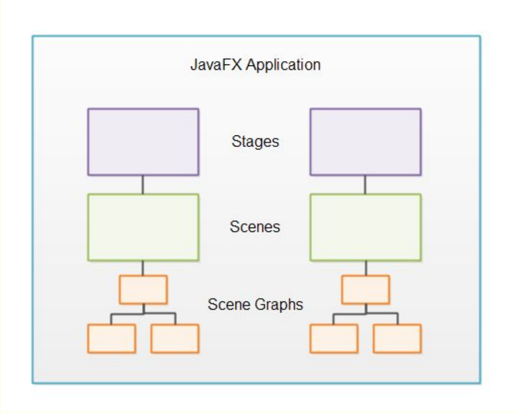
Stage



- outer frame (window)
- primary Stage object

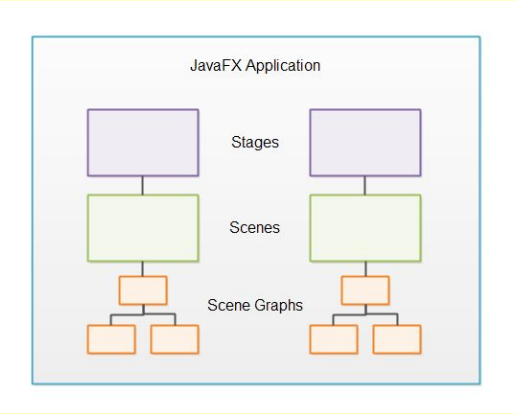
  created by the JavaFX runtime

Scene



- a stage can only show one scene at a time
- you can exchange scenes at runtime

Scene Graph



- Controls must be attached to scenes
- Components attached are called **nodes**
  - branch nodes (parent nodes)
  - leaf nodes

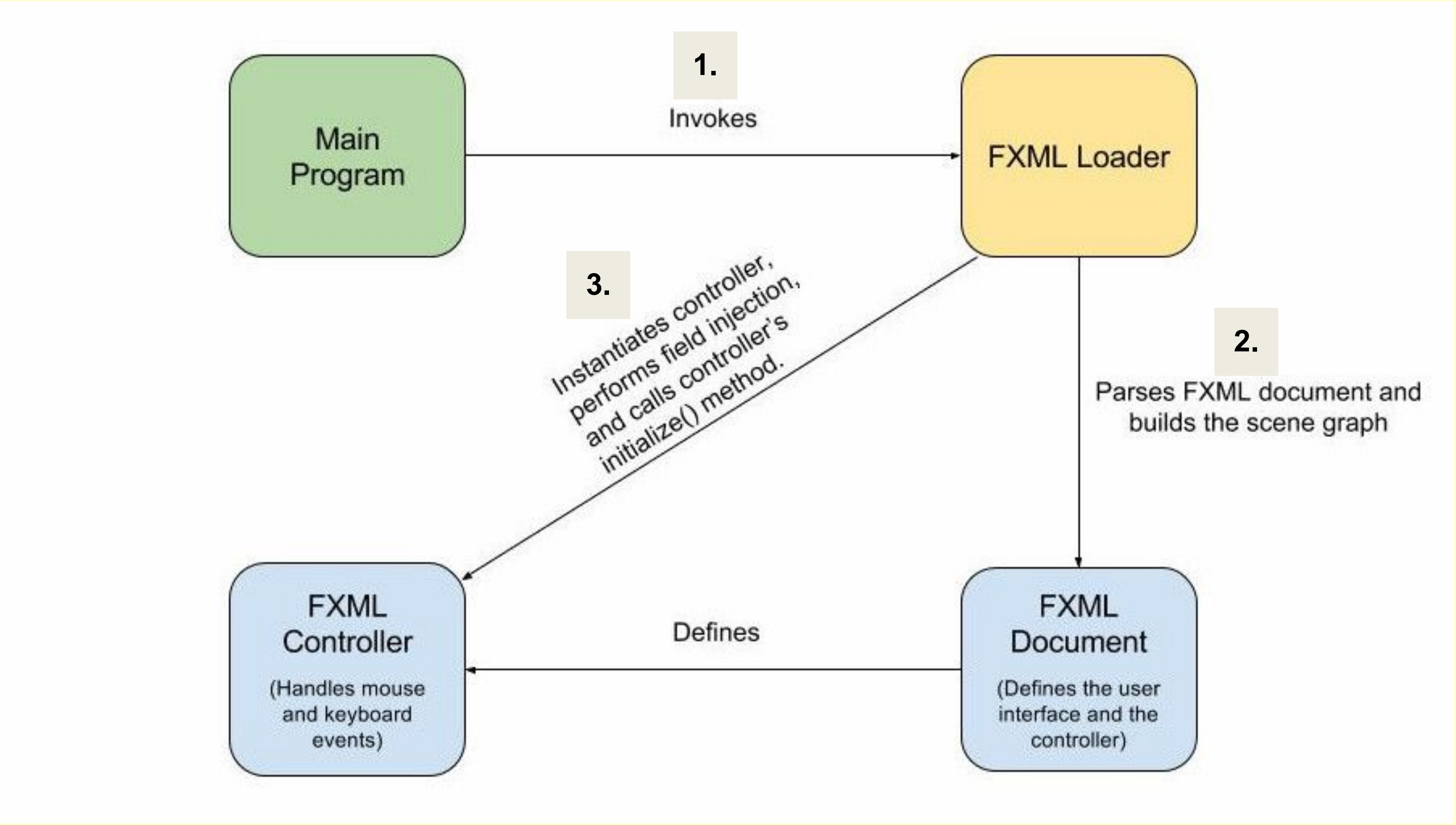
Your first JavaFX application

```
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root =
            FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("First App");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

JavaFX UI

1. Declarative UI - FXML
2. Programmatic UI - Java

1. Declarative UI



FXML - Adding UI elements

- top-level element (**layout**)
- children (controls)

```
<VBox>
  <children>
       <Label/>
       <TextField/>
       <Button/>
  </children>
</VBox>
```

FXMLLoader

```
@Override
public void start(Stage primaryStage) throws Exception{
    Parent root =
    FXMLLoader.load(getClass().getResource("sample.fxml"));
    primaryStage.setTitle("Hello World");
    primaryStage.setScene(new Scene(root, 300, 275));
    primaryStage.show();
}
```

FXML - Control properties

```
<TextField
    fx:id="inputText"
    prefWidth="100.0" />

<Button
    fx:id="okBtn"
    alignment="CENTER_RIGHT"
    onAction="#refresh"
    text="OK"
    textAlignment="CENTER" />
```

FXML - Event handling

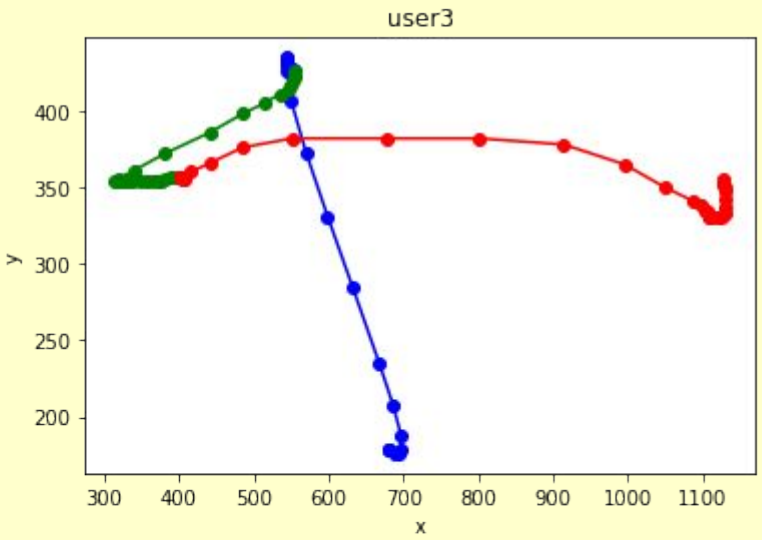
```
<TextField
    fx:id="inputText"
    prefWidth="100.0" />

<Button
    fx:id="okBtn"
    alignment="CENTER_RIGHT"
    onAction="#refresh"
    text="OK"
    textAlignment="CENTER" />
```

```
public class Controller {
  public void refresh(ActionEvent e){
    Button button = (Button)e.getSource();
    // ...
  }
}
```

ActionEvent

MouseLogger application

Create an application that logs the mouse events!



Event handling - Mouse Events (1)

```
<GridPane fx:controller="sample.Controller"
      xmlns:fx="http://javafx.com/fxml"
    alignment="center" hgap="10" vgap="10"

    onMouseMoved     = "#handleMouseMoved"
    onMousePressed   = "#handleMousePressed"
    onMouseReleased  = "#handleMouseReleased"
    onMouseDragged   = "#handleMouseDragged">
```

Event handling - Mouse Events (2)

```
public class Controller {
    private PrintStream out =
                     new PrintStream("mouse.csv");

public void handleMouseMoved(MouseEvent mouseEvent){
      out.println("MouseMove," +
             mouseEvent.getX()+","+mouseEvent.getY());
}
}
```

2. JavaFX - Programmatic UI

```
public GridPane createGridPane(){
    // ...
}

public void start(Stage primaryStage) throws Exception{
        primaryStage.setTitle("Data App");
        primaryStage.setScene(new Scene(createGridPane()));
        primaryStage.show();
}
```

2. JavaFX - Programmatic UI (cont)

```
public GridPane createGridPane(){
    GridPane gridPane = new GridPane();
    //…
    Button submitButton = new Button("Submit");
    // …
    gridPane.add(submitButton, 0, 3);

    submitButton.setOnAction(new EventHandler<ActionEvent>() {
       @Override
       public void handle(ActionEvent actionEvent) {
        // handle the event
        }
    });

}
```

JavaFX - Event handling

User interactions → Events

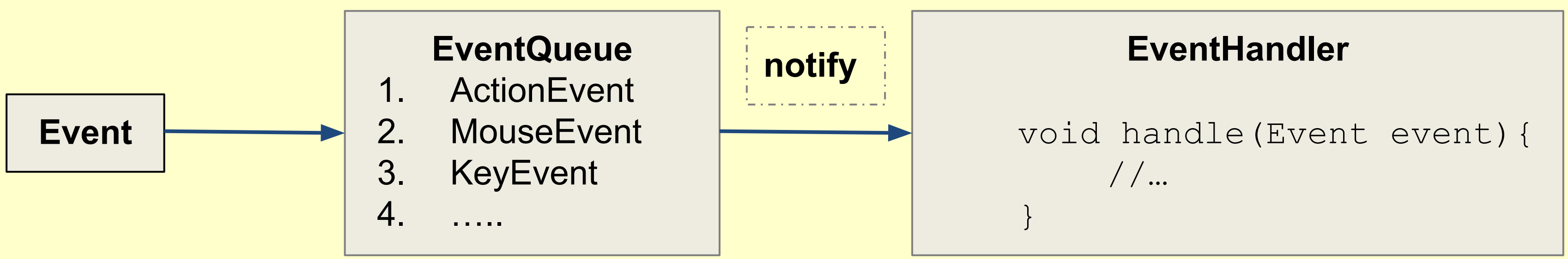
Types of events

- Foreground events
  - require direct interaction of a user
  - interactions with the UI
- Background events
  - operating system interruptions
  - timer expiry

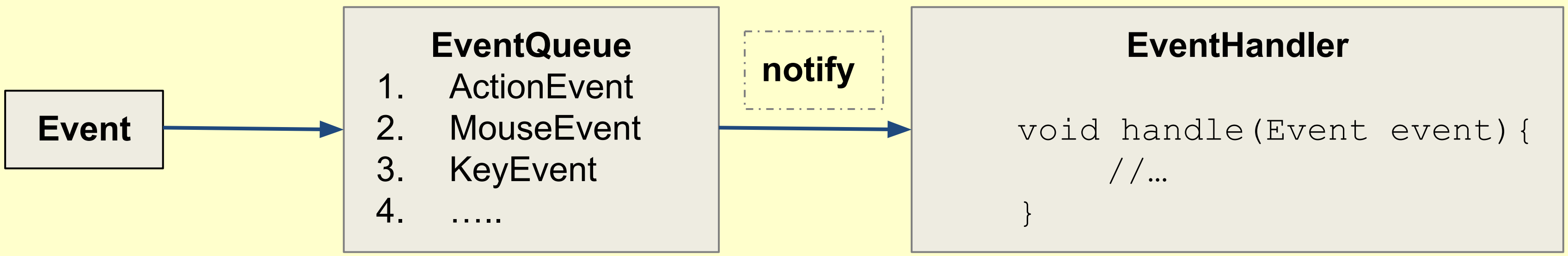
Event Driven Programming

EventDispatcher:

- receives events
- notifies interested objects



Event Driven programming (cont)



1. JavaFX UI → user clicks a button (event source)
2. System creates an ActionEvent (event queue)
3. If there is any event listener/handler registered to the button → it is notified → listener/handler runs event handler method

JavaFX Events

- Event (base class)
  - InputEvent
    - Mouse Event
    - Key Event
  - WindowEvent
  - ActionEvent
  - ...

Source of Events

A component (UI control/Node) can be a source of many kinds of events.

Node	Event Type
Button	ActionEvent
TextField	ActionEvent KeyEvent
Any kind of Node	MouseEvent

EventHandler

JavaFX - **one interface** for all kinds of event handlers

```
public interface EventHandler<T extends Event>{
   void handle(T event);
}
```

Example

ActionEvent

```
class ButtonHandler implements
                    EventHandler<ActionEvent>{
   public void handle(ActionEvent evt){
      //...
   }
}
```

Example (cont)

ButtonHandler usage

- use **addEventHandler**:

```
button.addEventHandler(
     ActionEvent.ALL, new ButtonHandler())
```

- use **setOnAction**:

```
button.setOnAction( new ButtonHandler() )
```

Ways to define event handlers

1. Define a class that **implements EventHandler** (*previous example*).

2. Write it as **anonymous class** (*if we need only once!*).

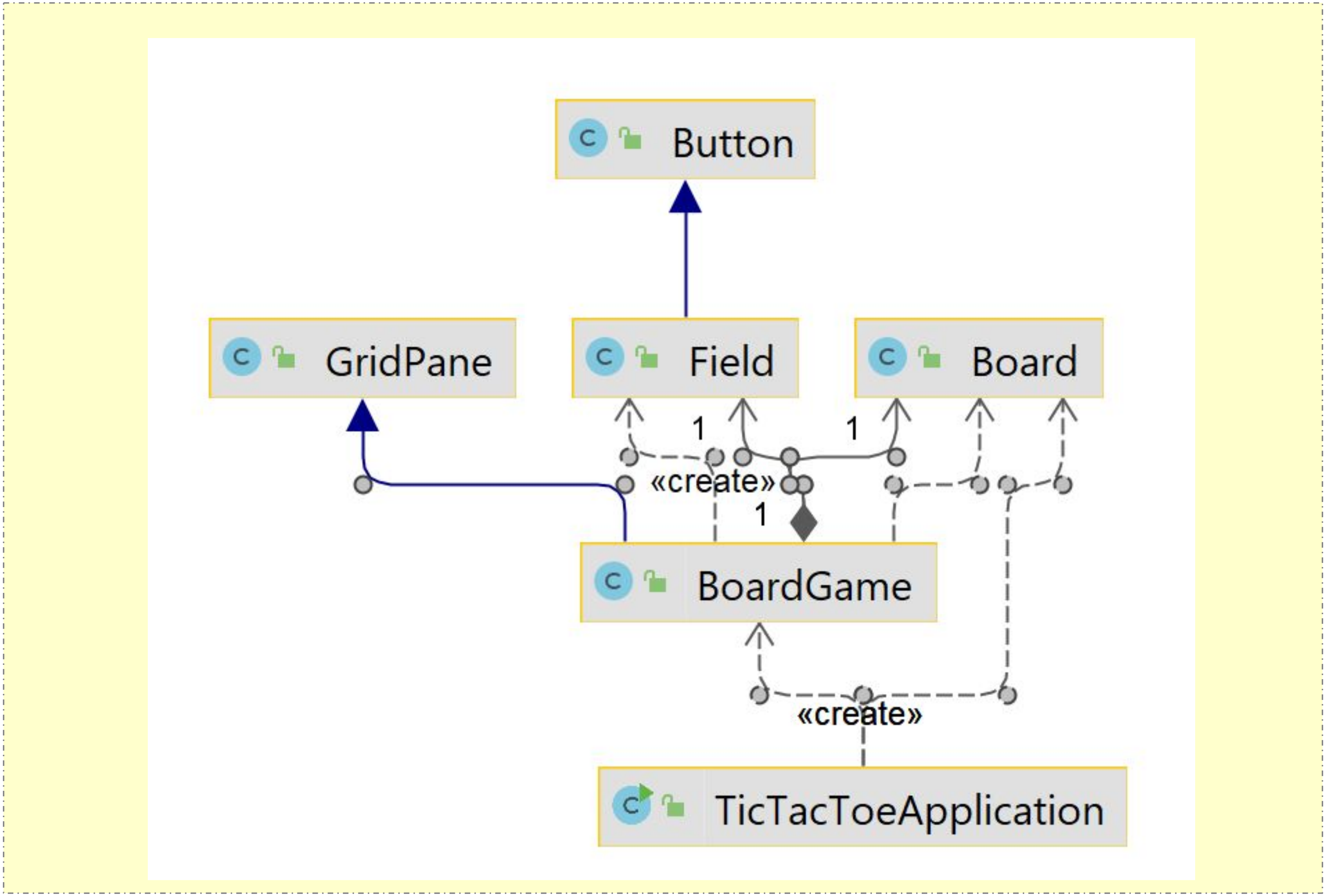
3. Write it as a **lambda expression** and use a reference variable to add it.

2. Anonymous class

```
submitButton.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent actionEvent) {
     // handle the event
     }
});
```

3. Lambda

```
submitButton.setOnAction(event-> {

    // statements to handle the event
    String firstname = firstnameTextField.getText();
    String lastname = lastnameTextField.getText();
    String email = emailTextField.getText();
    out.println( new Student(firstname, lastname, email));

    }
});
```


Button
GridPane
Field
Board
1
1
«create»
1
BoardGame
«create»
TicTacToeApplication

Module 11
Collections and Generics

Outline

- Data Structures
- Interfaces: **`Collection, List,Set,Map,...`**
- Implementations: **`ArrayList,HashSet,TreeMap,...`**
- Traversing collections
- Overriding **`equals`** and **`hashCode`**
- Sorting
- Problems

The Collections API

- What is?
  - Unified architecture
    - Interfaces – implementation-independence
    - Implementations – reusable data structures
    - Algorithms – reusable functionality
  - Best-known examples
    - C++ Standard Template Library (STL)
    - Smalltalk collections

The Collections API

- Benefits:
  - Reduces programming effort
  - Increases **performance**
    - High performance implementations of data structures
  - Fosters **software reuse**

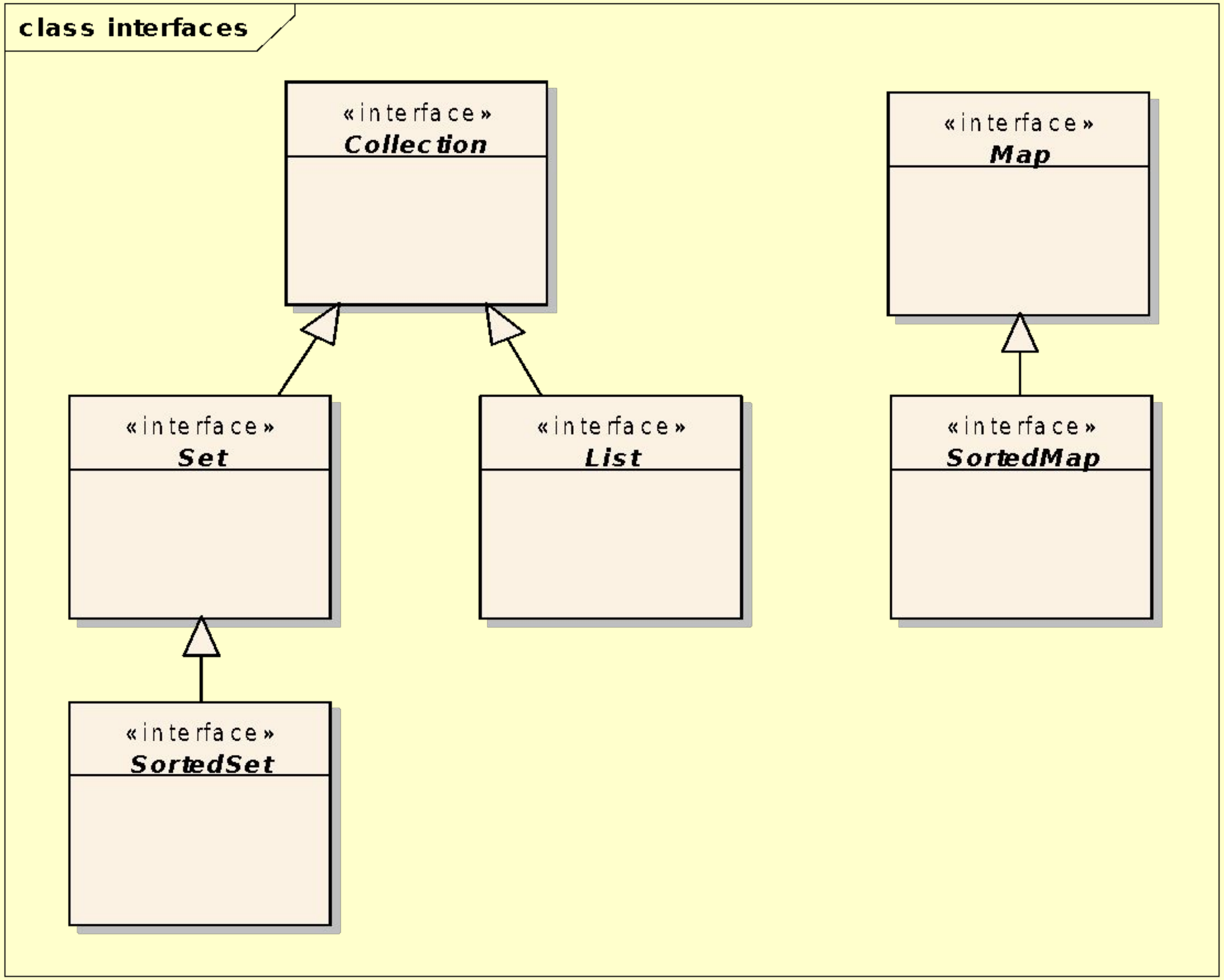
The Collections API

Design Goals

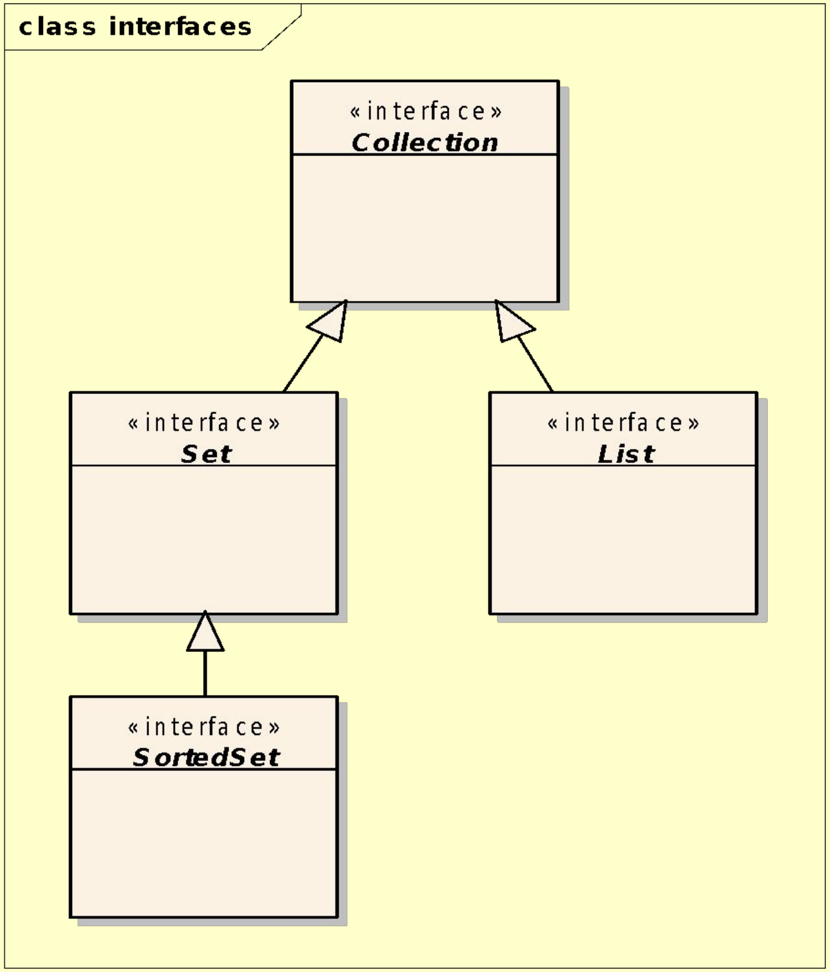
- Small and simple
- Powerful
- **Easily extensible**
- Compatible with preexisting collections
- Easy to use

The Collections API

Interfaces



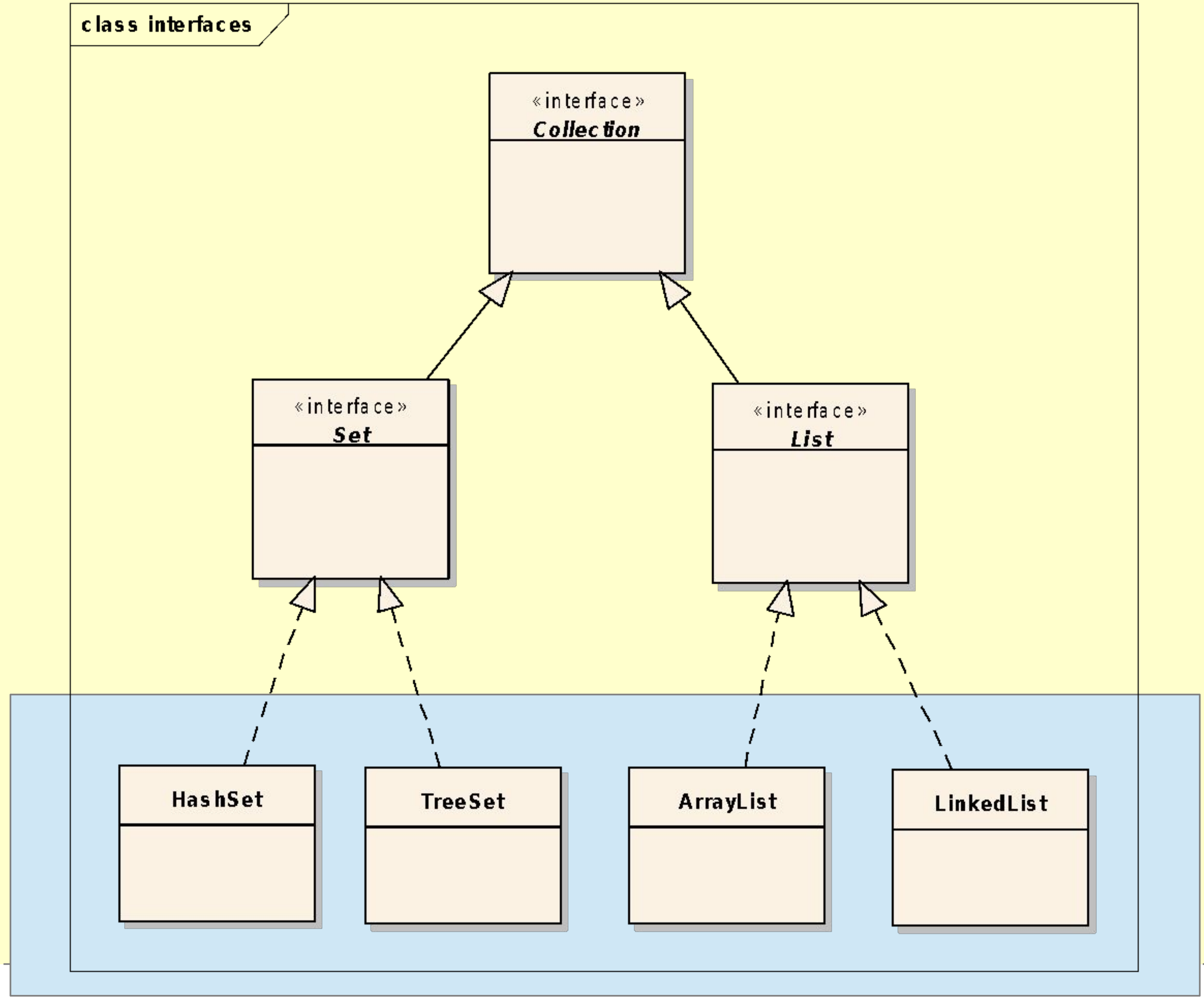
The `Collection` interface



Methods:

`add(T what): boolean`
`remove(T what): boolean`
`size(): int`
`contains(T what): boolean`
`containsAll(Collection c): boolean`
`equals(T what): boolean`
`iterator():` **`Iterator`**

Implementations

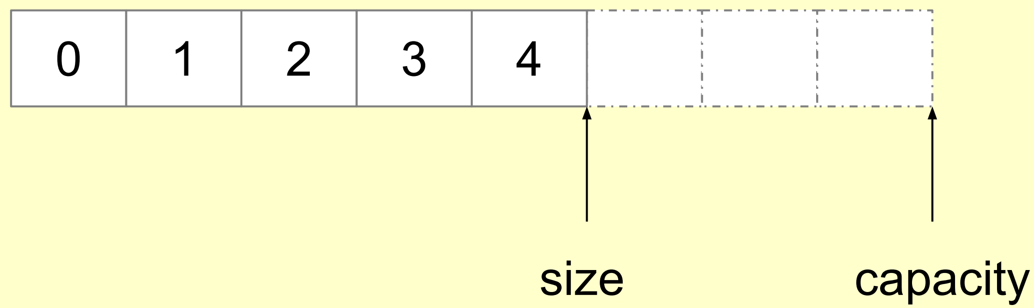


Implementations

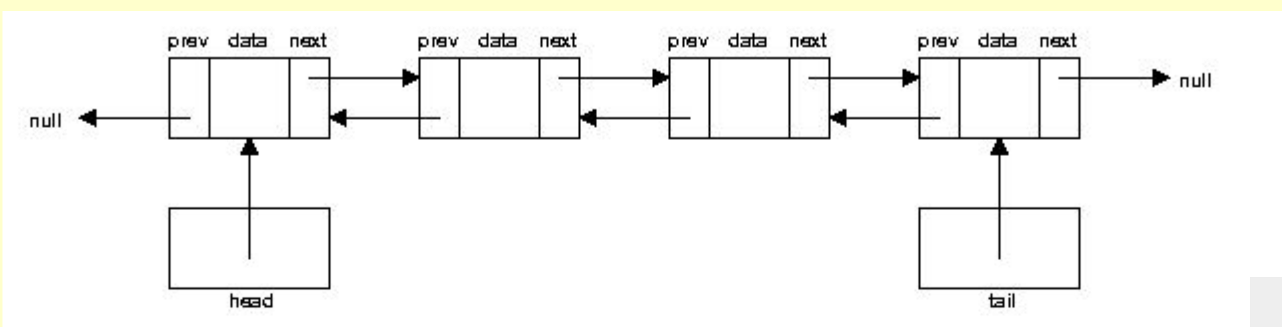
		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		Linked List
	Map	HashMap		TreeMap	

`List` implementations

ArrayList



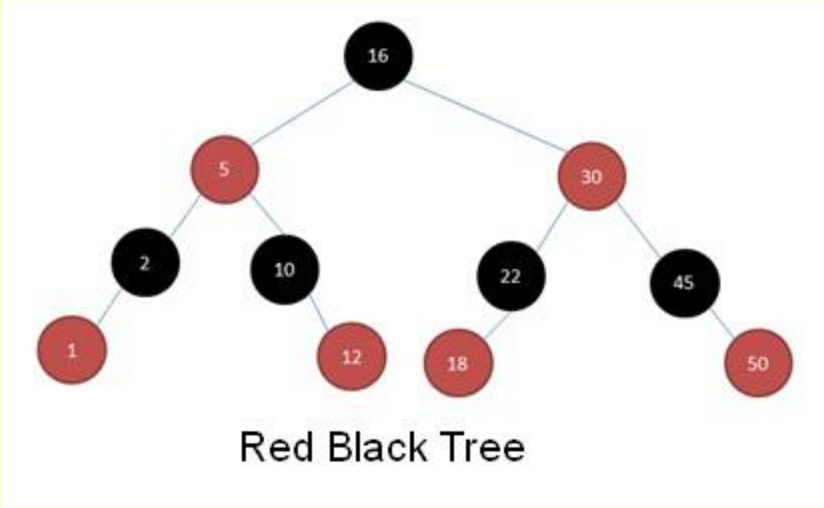
LinkedList



[Source]

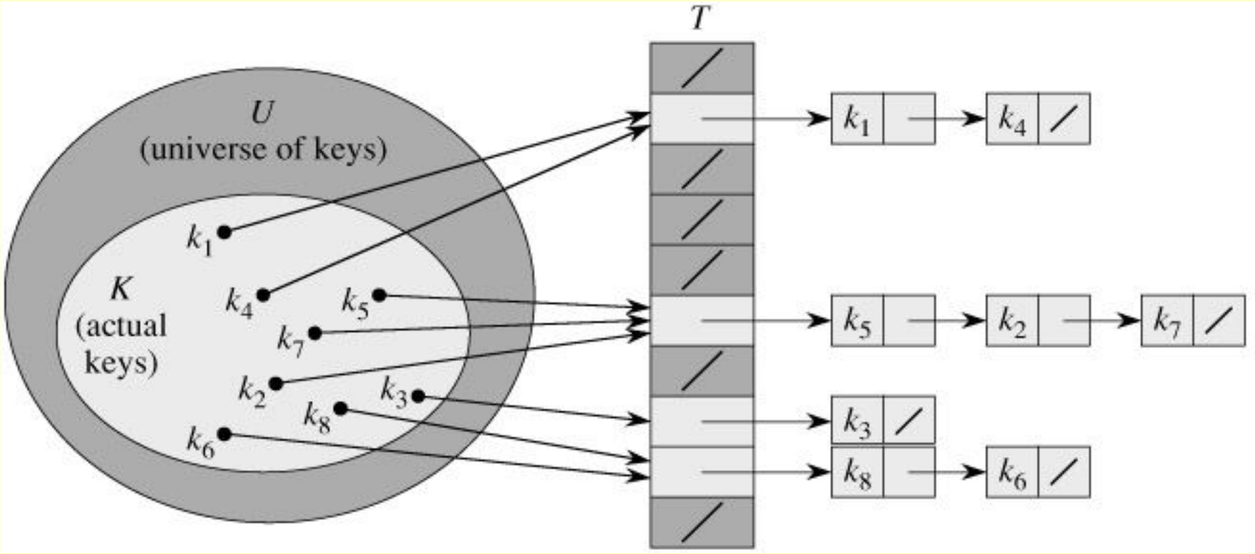
`Set` implementations

TreeSet



[Source]

HashSet



[Source]

Ordered vs. sorted collections

- **Ordered**
  - You can iterate through the collection in a specific (not random) order.
  - Each element has a previous and a next element (except the first and the last ones).
- **Sorted**
  - The order is determined according to some rule or rules (**sort order**).
  - Is a specific type of ordering
- **Collections**
  - **`HashSet:`** unordered and unsorted
  - **`List:`** ordered but unsorted
  - **`TreeSet:`** ordered and sorted

Complexities

	add (append)	get (position)	remove	contains
ArrayList	O(1)	O(1)	O(n)	O(n)
LinkedList	O(1)	O(n)	O(1)	O(n)
HashSet	O(1)*	-	O(1)*	O(1)*
TreeSet	O(log n)	-	O(log n)	O(log n)

* in the case of a **proper hash function**

Traversing Collections

There are 3 ways:

1) for-each
2) Iterator
3) Using aggregate operations (**since Java 8**)

Traversing Collections

(1) for-each

```
ArrayList list1 = new ArrayList();
…
for(Object o: list1){
   System.out.println(o);
}
--------------------------------------------
ArrayList<Person> list2 = new ArrayList<>();
…
for(Person p: list2){
 System.out.println(p);
}
```

Traversing Collections

(2) Iterator

```
package java.util;

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

Traversing Collections

(2) Iterator

```
ArrayList list1 = new ArrayList();
…
Iterator it1 = list1.iterator();
while(it1.hasNext()){
   System.out.println(it1.next());
}
-----------------------------------------
ArrayList<Person> list2 = new ArrayList<>();
…
Iterator<Person> it2 = list2.iterator();
while(it2.hasNext()){
   System.out.println(it2.next());
}
```

Traversing Collections

(2) Iterator

```
ArrayList list1 = new ArrayLi
…
Iterator it1 = list1.iterator
while(it1.hasNext()){
   System.out.println(it1.nex
}
----------------------------------------
ArrayList<Person> list2 = new ArrayList<>();
…
Iterator<Person> it2 = list2.iterator();
while(it2.hasNext()){
   System.out.println(it2.next());
}
```

An Iterator is an object

- **State:** represents a position in a collection
- **Behavior:** permits to step through the collection

Traversing Collections

(3) Using lambdas

Java 8

```
List<String> fruits = new ArrayList<>(
       Arrays.asList("apple", "pear", "grapes", "strawberry"));
fruits.forEach( e -> System.out.println(e));
```

Lambdas - anonymous functions

```
param -> expression
(param1, param2) -> expression
(param1, param2) -> {code block}
```

Traversing Collections

(3) Using aggregate operations

Java 8

```
TreeSet<String> dict = new TreeSet<>();
Scanner scanner = new Scanner( new File("dict.txt"));
while( scanner.hasNext()){
    dict.add( scanner.next());
}
System.out.println("SIZE: "+dict.size());
long counter = dict.stream()
                   .filter( e ->
                        e.startsWith("the"))
                   .count();
System.out.println("#words: "+counter);
```

Problems

Which data structure to use?

Problem:

Split a text file into words and print the **distinct** words in

1) Increasing order (alphabetically)
2) Decreasing order

Problems

Which data structure to use?

Problem:

Split a text file into words and print the **distinct** words in

1) Increasing order (alphabetically)
2) Decreasing order

Solutions:

1) `TreeSet<String>`
2) `TreeSet<String> (Comparator<String>)`

Problems

Decreasing Order

```
TreeSet<String> set = new TreeSet<>();
//...
TreeSet<String> rev = new TreeSet<>(
                      new Comparator<String>(){
   @Override
   public int compare(String o1, String o2) {
      return o2.compareTo(o1);
   }
});
rev.addAll( set );
```

Problem

Which data structure to use?

- **Problem:** Generate **2D points** having integer coordinates and print them in increasing order. Points should be **sorted** according to their distance to the origin.

Problem

2D Points

```
public class Point implements Comparable<Point>{
    public static final Point origin = new Point(0,0);

    private final int x, y;
    // constructor + getters
    public String toString(){ //...}
    public boolean equals(Object obj){ //...}
    public double distanceTo( Point point ){ //...}

    @Override
    public int compareTo(Point o) {
      return
        Double.compare(this.distanceTo(origin), o.distanceTo(origin));
    }
}
```

Problem

2D Points

Discussion!

```
public class Point implements Comparable<Point>{
    public static final Point origin = new Point(0,0);
```

```
TreeSet<Point> points1 = new TreeSet<>();
// OR
ArrayList<Point> points2 = new ArrayList<>();
Collections.sort(points2);
```

```
    public double distanceTo( Point point ){ //...}

    @Override
    public int compareTo(Point o) {
        return
        Double.compare(this.distanceTo(origin), o.distanceTo(origin));

    }
}
```

Problem

Generate randomly `N = 1.000.000` (one million) **distinct** bidimensional points (x, y) having positive integer coordinates (`0 =< x <= M, 0 =< y <= M, M = 1.000.000`).

Requirements:

- Optimal solution is required.
- Print the number of duplicates generated.

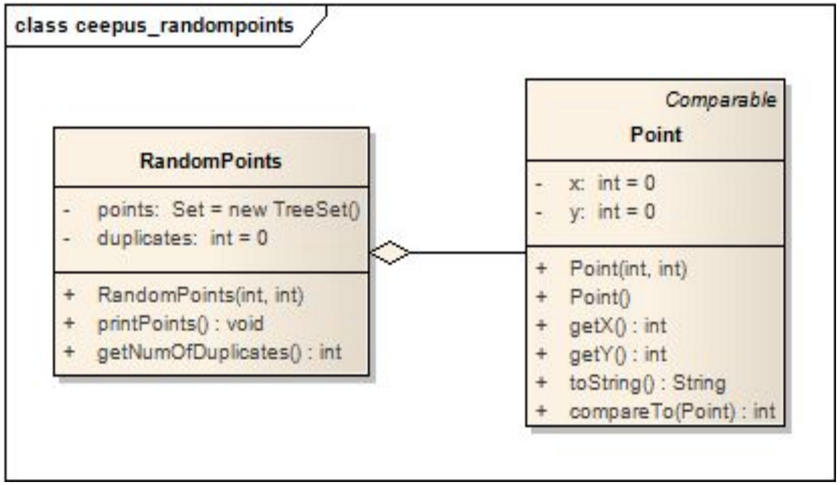
Which collection to use?

Hint: Finding an existing element must be fast.

Problem

1. solution - TreeSet

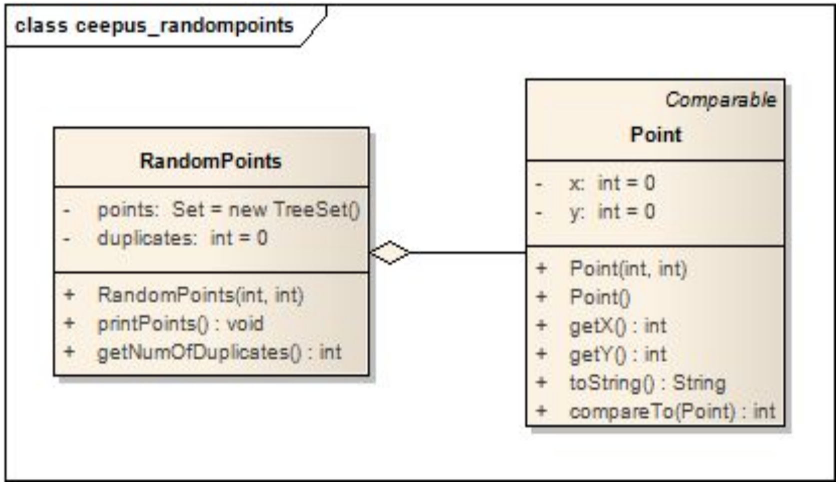
```
public class Point implements
                    Comparable<Point> {
  ...
   @Override
   public int compareTo(Point o) {
        if( o == null ) throw
           new NullPointerException();
        if (this.x == o.x &&
            this.y == o.y){
            return 0;
        }
        if(this.x == o.x){
           return
             Integer.compare(this.y, o.y);
        } else{
             Integer.compare(this.x, o.x);
        }
  }
}
```



Problem

1. solution - TreeSet

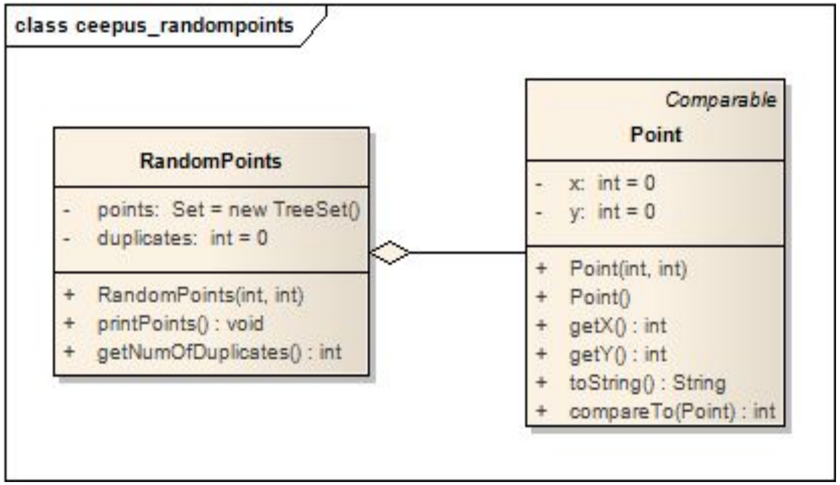
```
public class RandomPoints {
    private TreeSet<Point> points =
                   new TreeSet<Point>();
    private int duplicates = 0;

    public RandomPoints( int size,
                            int interval){
        int counter = 0;
        Random rand = new Random(0);
        while( counter < size ){
            int x =
          Math.abs(rand.nextInt() % interval);
            int y =
          Math.abs(rand.nextInt() % interval);
            Point p = new Point(x,y);
            if( points.contains( p )){
               ++duplicates;
               continue;
            }
            ++counter;
            points.add(p);
        }
    }
    …
}
```



Problem

1. solution - TreeSet

```
public class RandomPoints {
    private TreeSet<Point> points =
                   new TreeSet<Point>();
    private int duplicates = 0;

    public RandomPoints( int size,
                         int interval){
        int counter = 0;
        Random rand = new Random(0);
        while( counter < size ){
            int x =
          Math.abs(rand.nextInt() % interval);
            int y =
          Math.abs(rand.nextInt() % interval);
            Point p = new Point(x,y);
            if( points.contains( p )){
               ++duplicates;
               continue;
            }
            ++counter;
            points.add(p);
        }
    }
    …
}
```



TreeSet

- Finding an element: O(log n)

Implementation

Random number generator: seed = 0

N = 1.000.000

M = 10.000

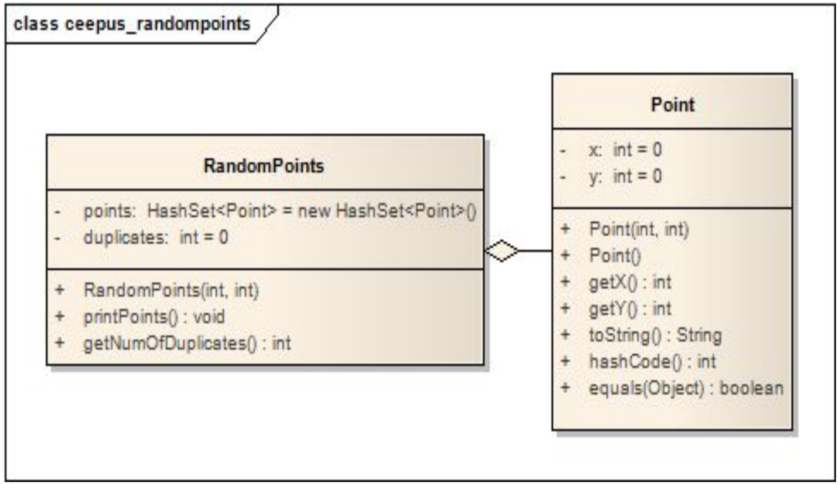
Duplicates: 4976

Time: **approx. 3s**

Problem

2. solution - HashSet

```
@Override
public int hashCode() {
    int hash = (x *31)^ y;
    return hash;
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
            return false;
    }
    final Point other = (Point) obj;
    if (this.x != other.x) {
            return false;
    }
    if (this.y != other.y) {
            return false;
    }
    return true;
}
```



HashSet

- Finding an element: O(1)

Implementation

Random number generator: seed = 0

N = 1.000.000

M = 10.000

Duplicates: 4976

Time: **approx. 1s**

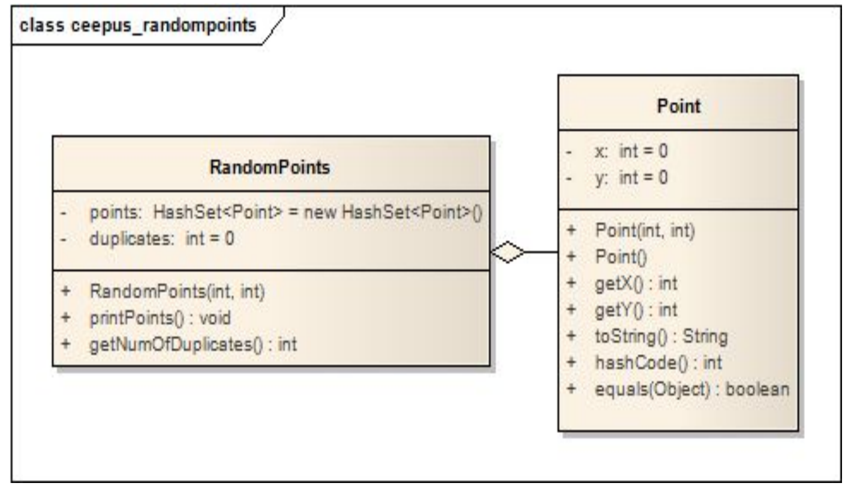
Problem

2. solution - HashSet

```
@Override
public int hashCode() {
    int hash = (x *31)^ y;
   return hash;
}

@Override
public boolean equals(Object obj) {
   if (obj == null) {
     return false;
   }
   if (getClass() != ob  .getClass()) {
           return false
   }
   final Point other =    oint) obj;
   if (this.x != other.    {
           return false
   }
   if (this.y != other.
           return false
   }
   return true;
}
```

What happens if we don't override equals?

How many duplicates?



HashSet

- Finding an element: O(1)

Implementation

Random number generator: seed = 0

N = 1.000.000

M = 10.000

Duplicates: 4976

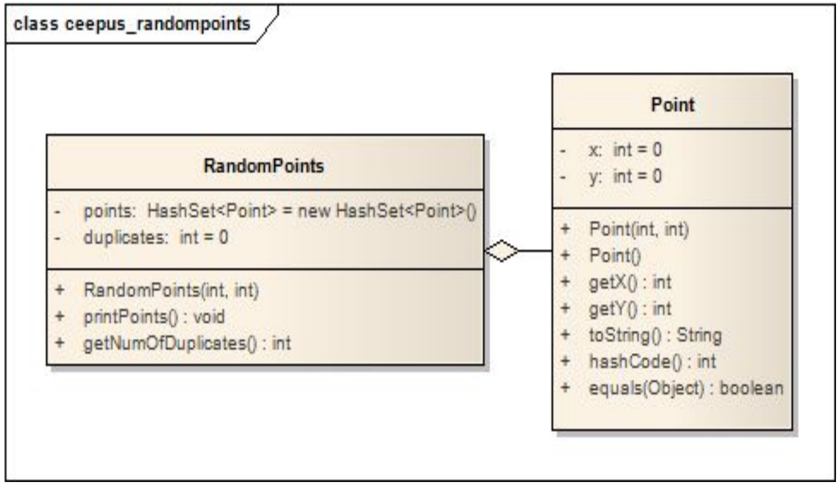
Time: **approx. 1s**

Problem

2. solution - HashSet

```
@Override
public int hashCode() {
    int hash = 1;
    return hash;
}

@Override
public boolean equa s(Object obj) {
  if (obj == null)
    return false;
  }
  if (getClass() != o   getClass()) {
          return false
  }
  final Point other = (   nt) obj;
  if (this.x != other.x)
          return false;
  }
  if (this.y != other.y)
          return false;
  }
  return true;
}
```

What happens?



Problem

2. solution - HashSet

The **`hashCode()`** contract:

- each time invoked on the same object must return the same value (consistent, can't be random)
- if **`x.equals(y) == true`**, then
  `x.hashCode() == y.hashCode()` must be true
- It is legal to have the same hashcode for two distinct objects (collision)

Problem
3. solution

Which collection to use if `M = 2000`

Hint: Which is the fastest access time of an element in a collection?

Problem

3. solution

Which collection to use if `M = 2000`

Hint: Which is the fastest access time of an element in a collection?

```
private boolean exists[ ][ ] = new boolean[ M ][ M ];
```

```
public RandomPoints( int size, int interval){
   int counter = 0;
   Random rand = new Random(0);
   while( counter < size ){
      int x = Math.abs(rand.nextInt() % interval);
      int y = Math.abs(rand.nextInt() % interval);
      Point p = new Point(x,y);
      if( exists[ x ][y ]){
        ++duplicates;
        continue;
      }
      ++counter;
      exists[ x ][ y ] = true;
   }
}
```

	0	1	2	3	4	5	6	7
0								
1				T				
2								
3					T			
4		T						
5								
6								
7								

Problem

3. solution

Which collection to use if `M = 2000`

Hint: Which is the fastest access time of an element in a collection?

```
private boolean exists

public RandomPoints( int s
  int counter = 0;
  Random rand = new Random
  while( counter < size ){
    int x = Math.abs(rand.
    int y = Math.abs(rand.
    Point p = new Point(x,
    if( exists[ x ][y ]){
     ++duplicates;
     continue;
    }
    ++counter;
    exists[ x ][ y ] = true;
  }
}
```

Bidimensional array of booleans

- Finding an element: O(1)

Implementation

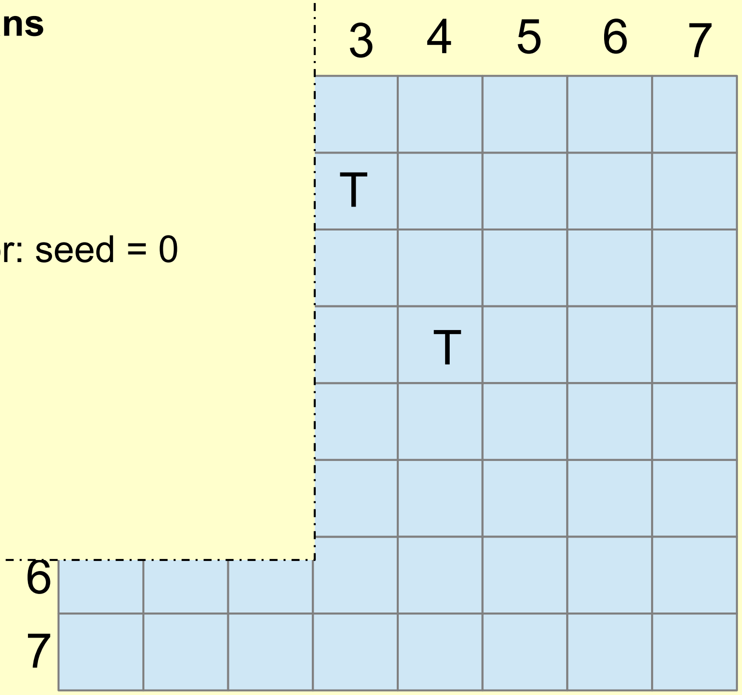
Random number generator: seed = 0

N = 1.000.000

M = **2000**

Duplicates: 150002

Time: **approx. 0.2 s**



Map

Interface

```
interface Map<K, V>
```

- **K** – Key type
- **V** – Value type

interface **Map.Entry<K,V>**
(Key, Value) pair

Maps keys to values.

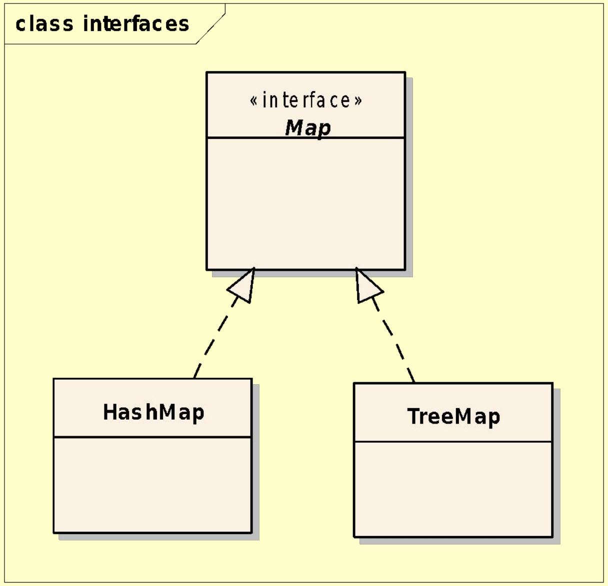
Examples:

Key: country, **Value**: capital city

- Slovenia → Ljubljana
- Austria → Vienna
- Hungary → Budapest
- Romania → Bucharest

Map

Implementations



HashMap: unordered, no duplicates

TreeMap: ordered by key, no duplicates

	get	put	remove
TreeMap	O(log n)	O(log n)	O(log n)
HashMap	O(1)*	O(1)*	O(1)*
* in the case of a proper hash function			

Map

Important methods

```
Map<K, V>

   V put(K key, V value)
   V get(Object key)
   V remove(Object key)

   Set<K> keySet()
   Collection<V> values()
   Set<Map.Entry<K,V>> entrySet()
```

Map

Print entries of a map (1)

```
Map<String, Counter> map = new TreeMap<>();
// fill the map

for(Map.Entry<String, Counter> e: map){
  System.out.println(e.getKey()+":"+e.getValue());
}
```

Map

Print entries of a map (2)

```
Map<String, Counter> map = new TreeMap<>();
// fill the map

for(String key: map.keySet()){
  System.out.println(key + ":" + map.get(key));
}
```

Map

Print entries of a map (3)

```
Map<String, Counter> map = new TreeMap<>();
// fill the map

map.forEach((key, value) -> {
     System.out.println(key + ": " + value);
});
```

Problem

Which data structure to use?

Problem:

Compute the word frequencies in a text. Print the words and their frequencies:

1) alphabetically,
2) in decreasing frequency order.

Problem

Solution (1) alphabetically

```
class MyLong {
    private long value;
    public MyLong(int value) { this.value = value;}
    public long getValue() { return value;}
    public void setValue(long value) { this.value = value;}
    public void increment() { ++value;}
}

//...
TreeMap<String, MyLong> frequency = new TreeMap<>();
```

Problem

Solution (2) decreasing frequency order

```
class Pair {
     private String word;
     private long fr;
     // constructor + get and set methods
}


ArrayList<Pair> list = new ArrayList<Pair>();
for (String key : frequency.keySet()) {
   long value = frequency.get(key).getValue();
   list.add(new Pair(key, value));
}
Collections.sort(list, new Comparator<Pair>() {
   @Override
   public int compare(Pair o1, Pair o2) {
      return  Integer.compare(o2.getFr(), o1.getFr());
   }
});
```

Problem

Which data structure to use?

Problem:

Find the anagrams in a text file!

Problem

Which data structure to use?

Problem: Find the anagrams in a text file!

Solution:

- Split the text into words
- Alphabetize the word
  - sent → enst
  - nest → enst
  - tens → enst
- `Map<String, List<String> >` vs. `Map<String, Set<String> >`
  - Key: alphabetized word → `String`
  - Value: words → `List<String>` or `Set<String>`

Problem

Anagrams

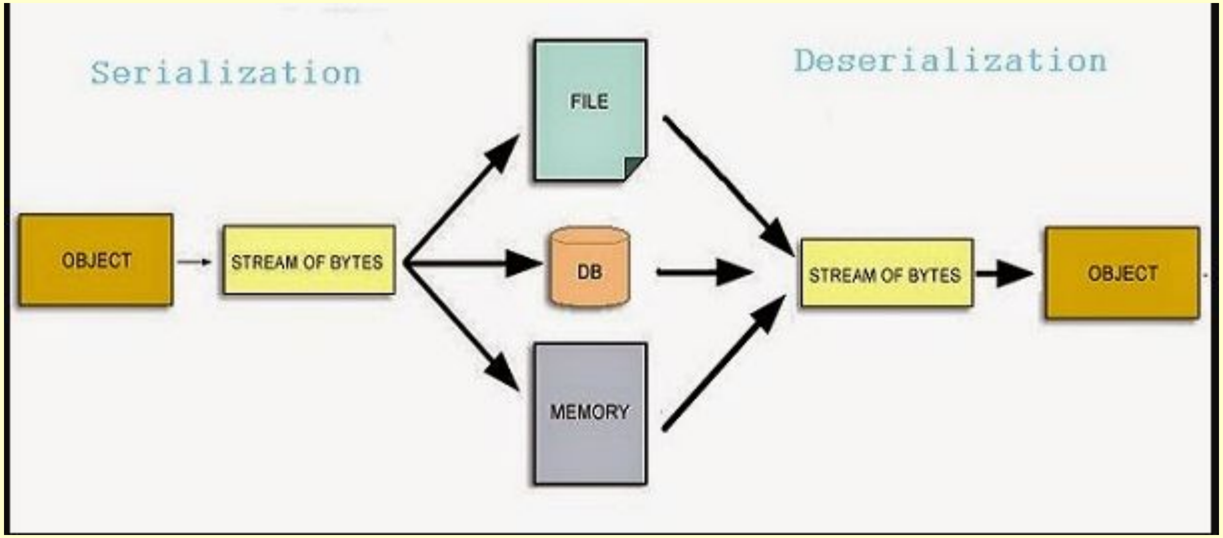
```
Map<String, Set<String> > groups = new HashMap<>();
//...

String word = cleanWord(word);
String key = alphabetize(word);
// Find the key
Set<String> group = groups.get(key);
if (group == null) {
     Set<String> newGroup = new HashSet<String>();
     newGroup.add(word);
     groups.put(key, newGroup);
} else{
    group.add(word);
}
```

Problem

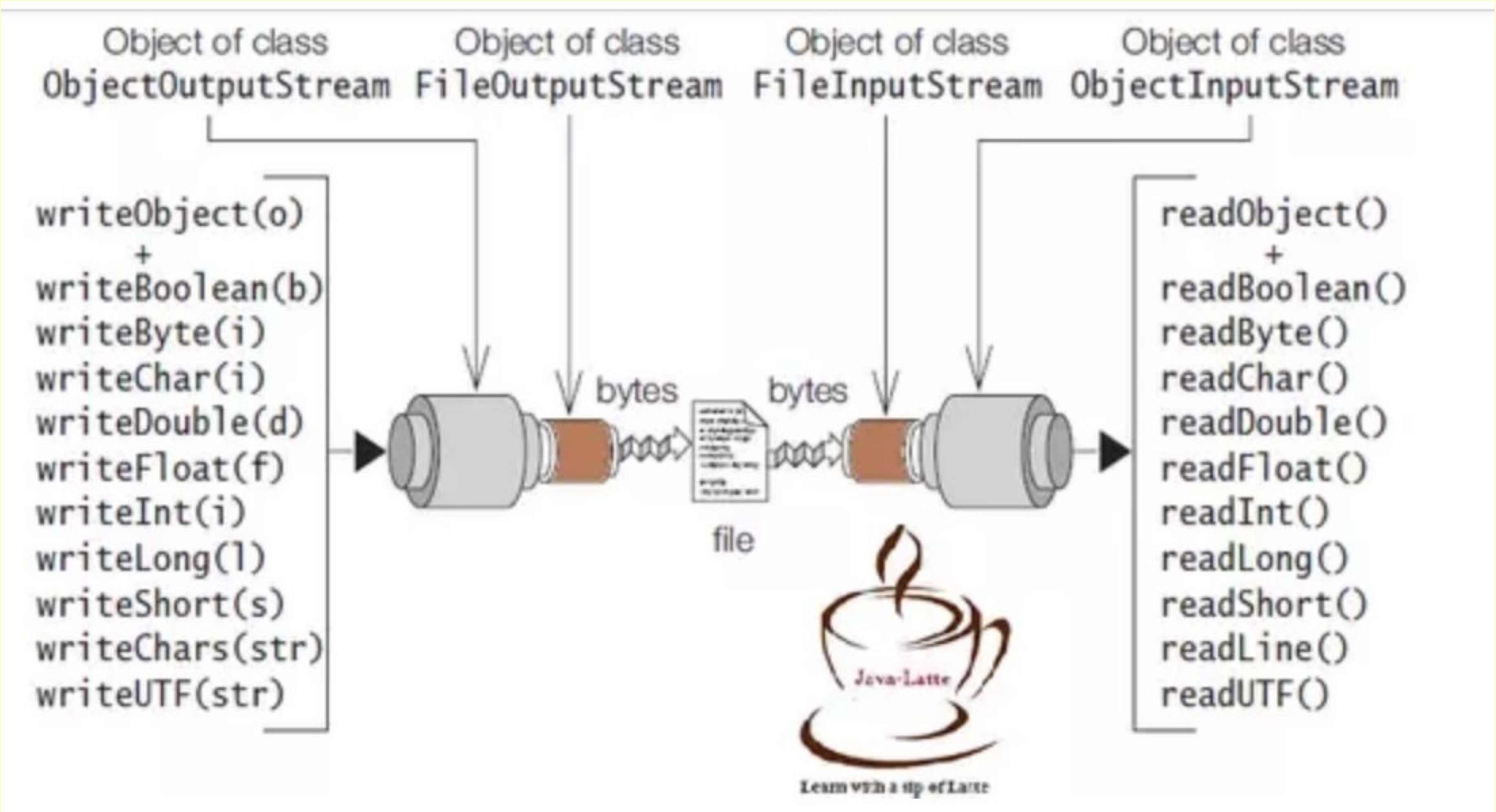
Anagrams

```
Map<String, Set<String> > groups = new HashMap<>();
//...

private void printGroups(int size) {
     for (String key : groups.keySet()) {
         Collection<String> group = groups.get(key);
         if (group.size() == size) {
         System.out.print("Key: " + key + " --> ");
            for (String word : group) {
                  System.out.print(word + " ");
            }
            System.out.println();
         }
     }
}
```

Module 12
Serialization



https://krishankantsinghal.medium.com/serialization-and-deserialization-5046c958c317



https://krishankantsinghal.medium.com/serialization-and-deserialization-5046c958c317

Rules

1. If a **parent class** has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.
2. Only **non-static data members** are saved via Serialization process.
3. **Static data members and transient data** members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.
4. **Constructor of object** is never called when an object is deserialized.
5. **Associated objects** must be implementing Serializable interface.

SerialVersionUID

1. Declared explicitly in the class
2. Calculated by the serialization runtime

Example

```
public class Student implements Serializable{
   private final String firstname;
   private final String lastname;
   private transient String password;
    // ...
}
```

Example (cont)

```
Student student1 = new Student("John", "Black");
// save the object to file
try (ObjectOutputStream out = new ObjectOutputStream( new
FileOutputStream("student.ser"))){
   out.writeObject(student1);
} catch (Exception e) {
   e.printStackTrace();
}
// read the object from file
try(ObjectInputStream in = new ObjectInputStream( new
FileInputStream("student.ser"))){
   student1 = (Student) in. readObject();
   System.out.println(student1);
   System.out.println("Counter: " +
                      Student.getCounter());

} catch (Exception e) {
   e.printStackTrace();
}
```