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*

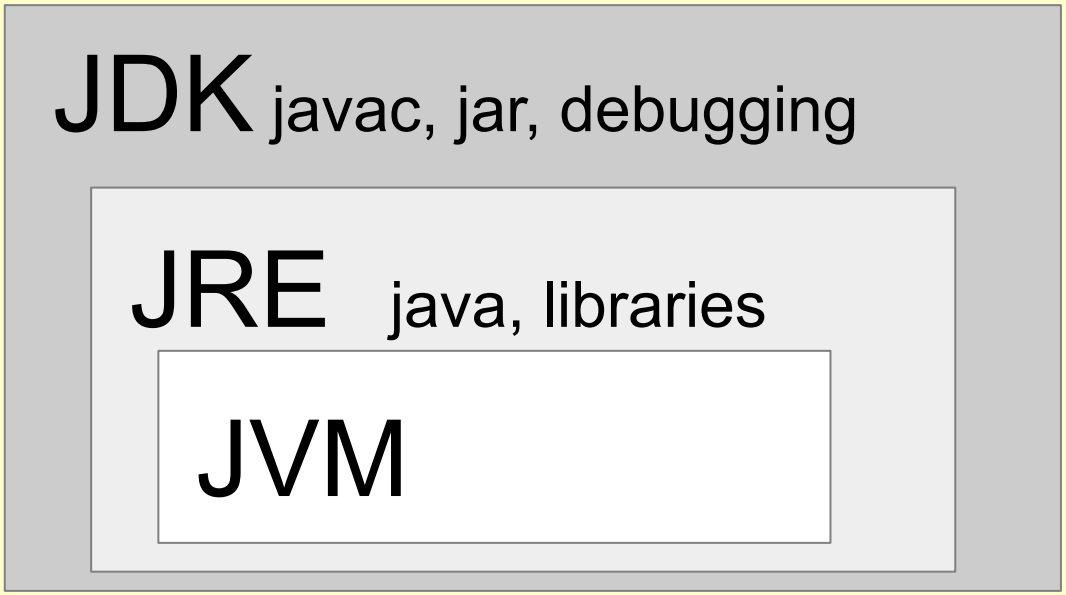
Short History

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

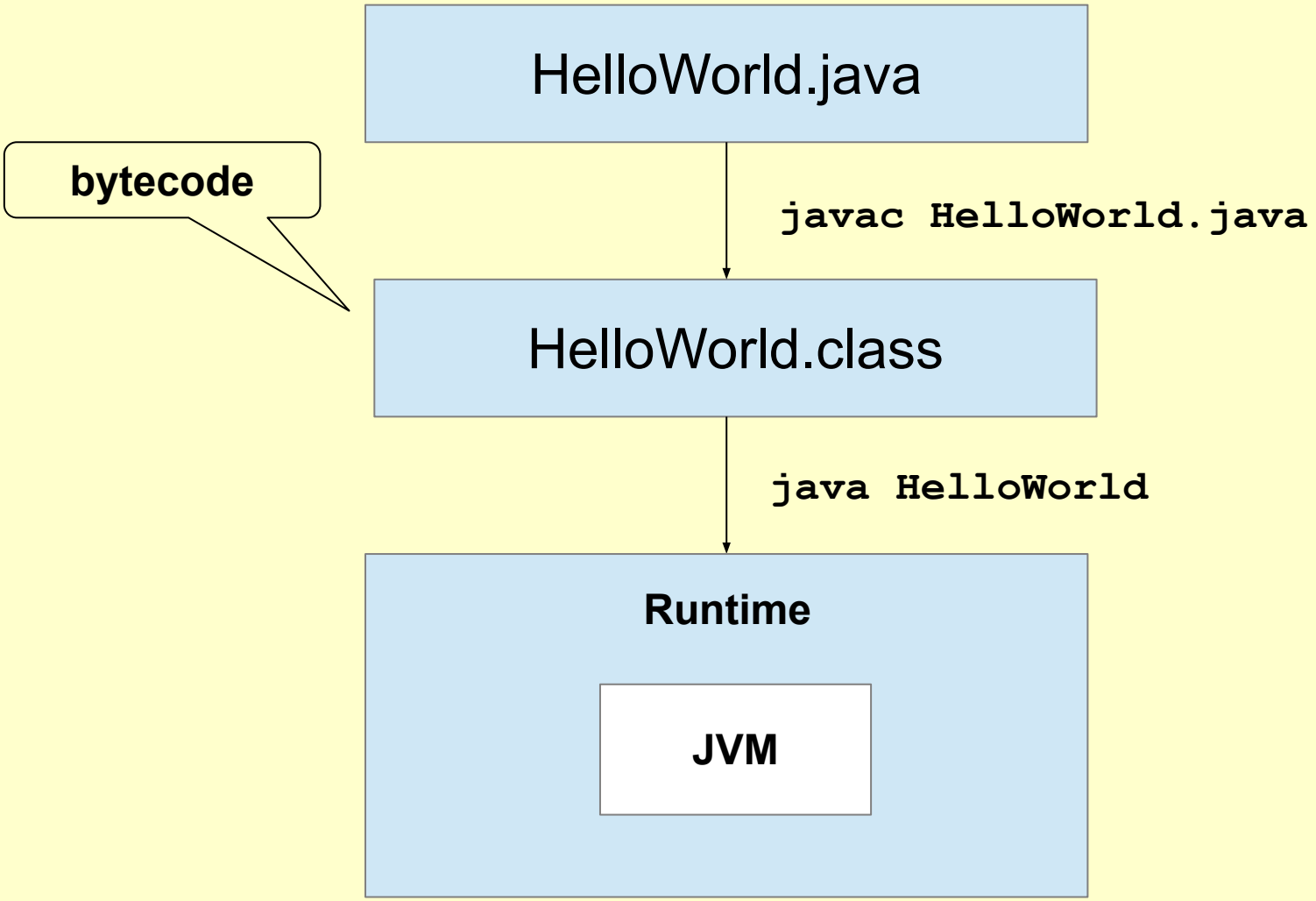
1. Write the source code: HelloWorld.java

```
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
```

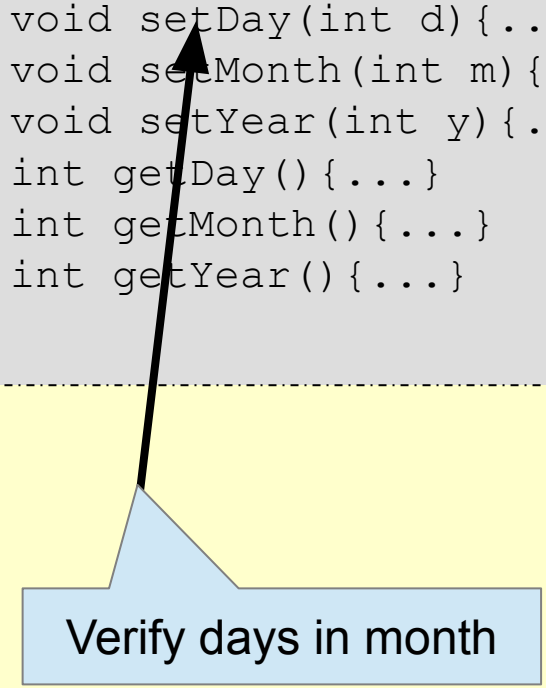
Information Hiding

- **The solution:**

Client code must use setters and getters to access internal data

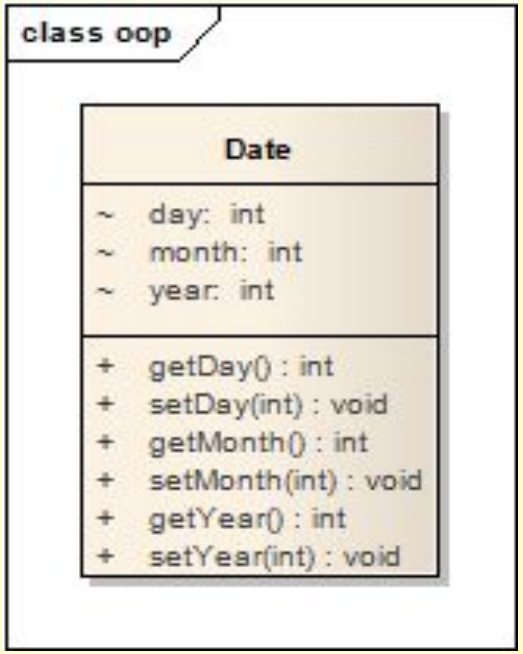
```
// Java language
public class Date {
   private int year, month, day;
   public void setDay(int d){..}
   public void setMonth(int m){..}
   public void setYear(int y){..}
   public int getDay(){...}
   public int getMonth(){...}
   public int getYear(){...}
}
```

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



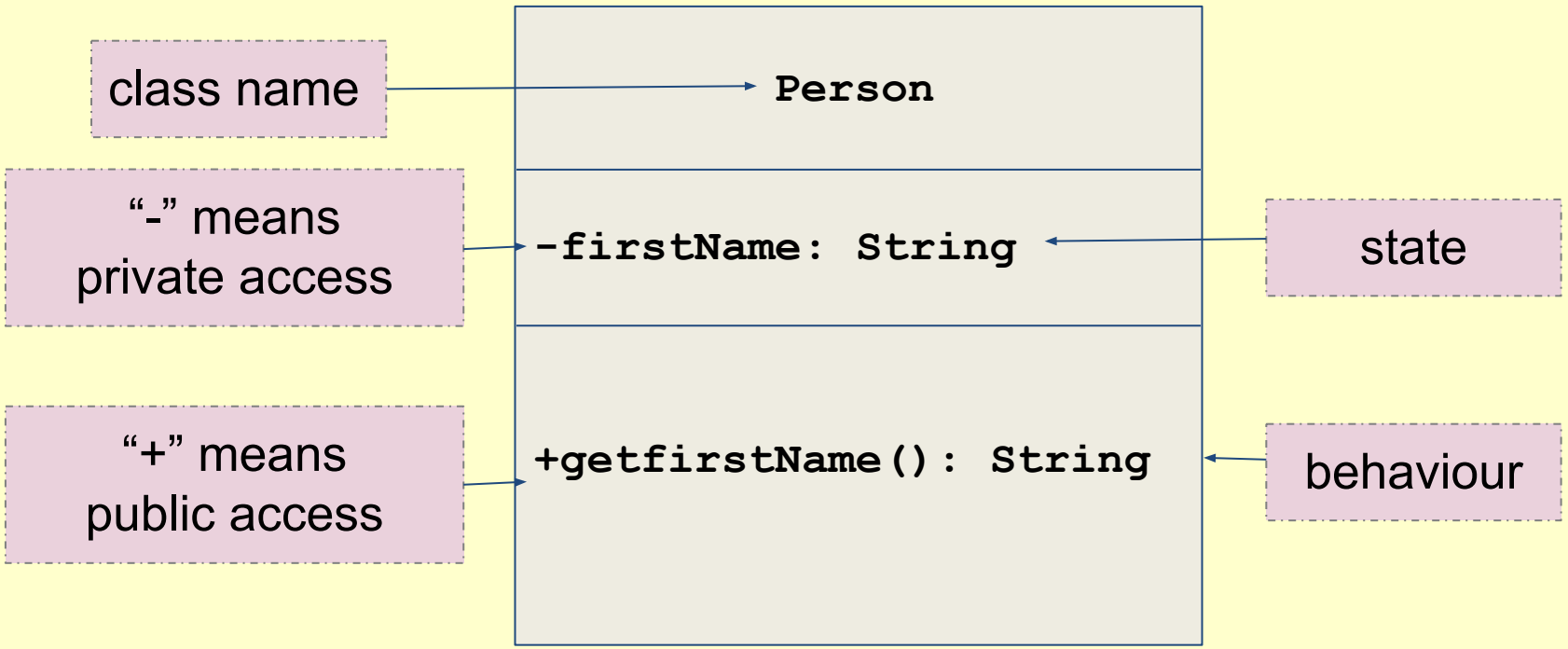
Encapsulation

- **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>*
}
```

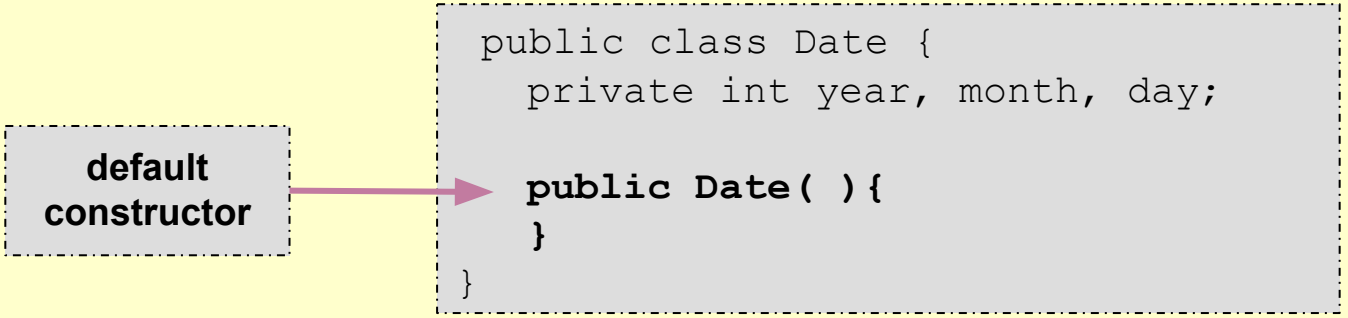
```
public class Date {
  private int year, month, day;

  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



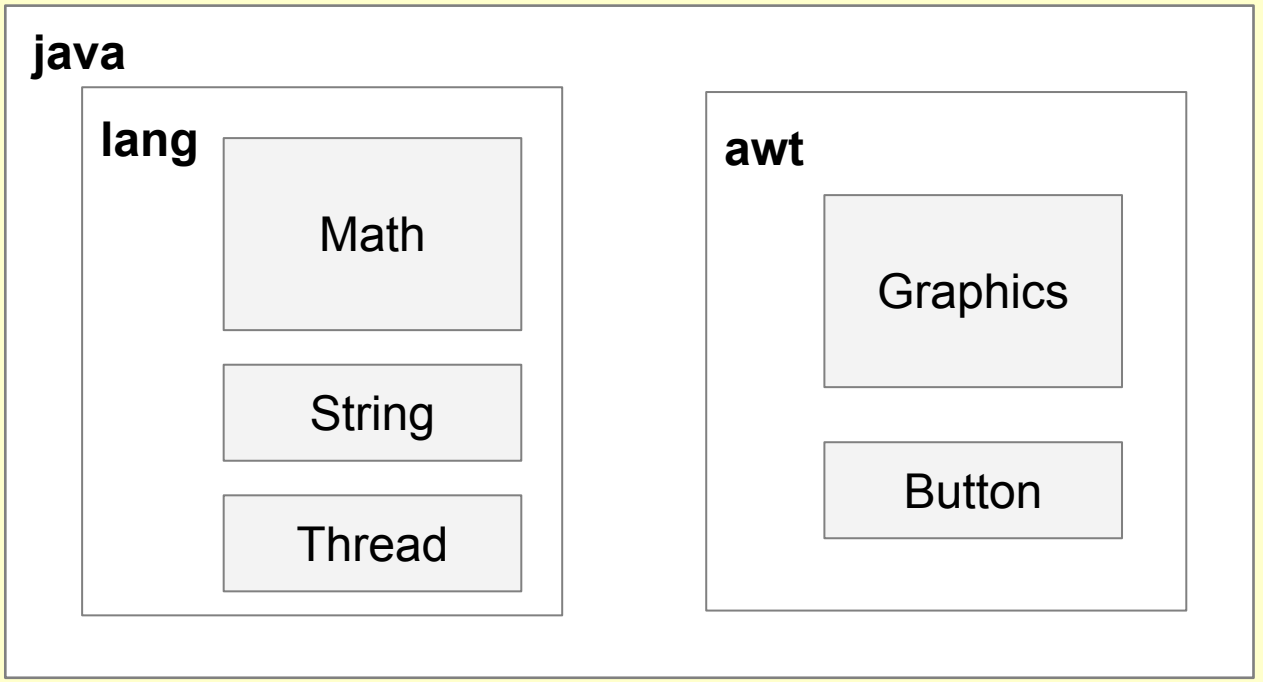
Objects

- 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

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

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

```
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);
```

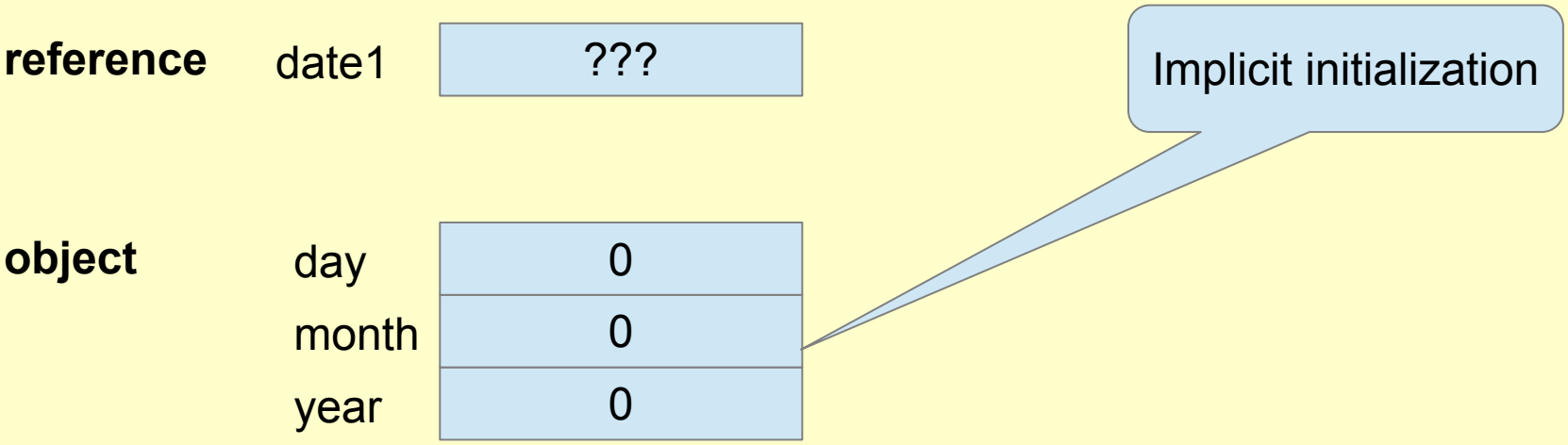
1) Memory is allocated for the object
2) Explicit attribute initialization is performed
3) A constructor is executed
4) The **object reference** is returned by the `new` operator

```
date1 =  object reference
```

5) The reference is assigned to a variable

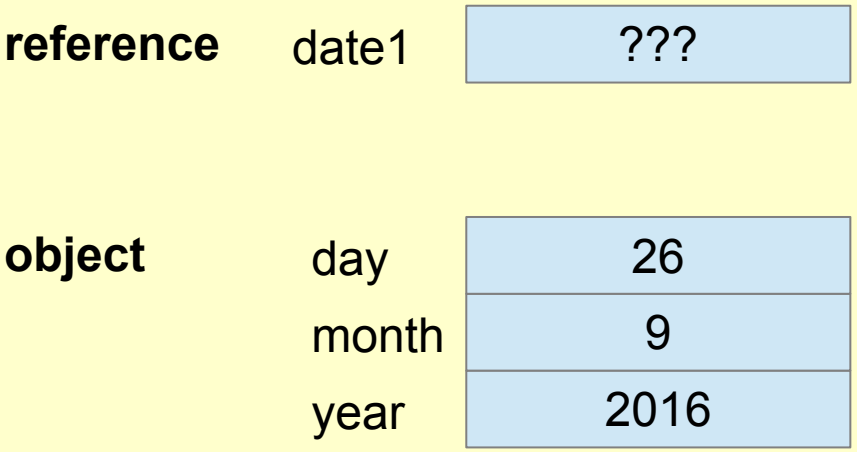
(1) Memory is allocated for the object

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



(2) Explicit Attribute Initialization

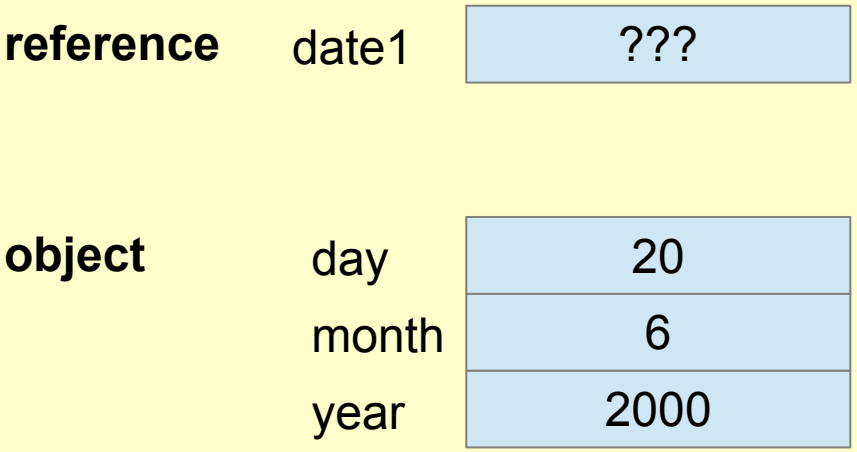
```
MyDate date1 = new MyDate(20, 6, 2000);
```




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

}
```

(3) Executing the constructor

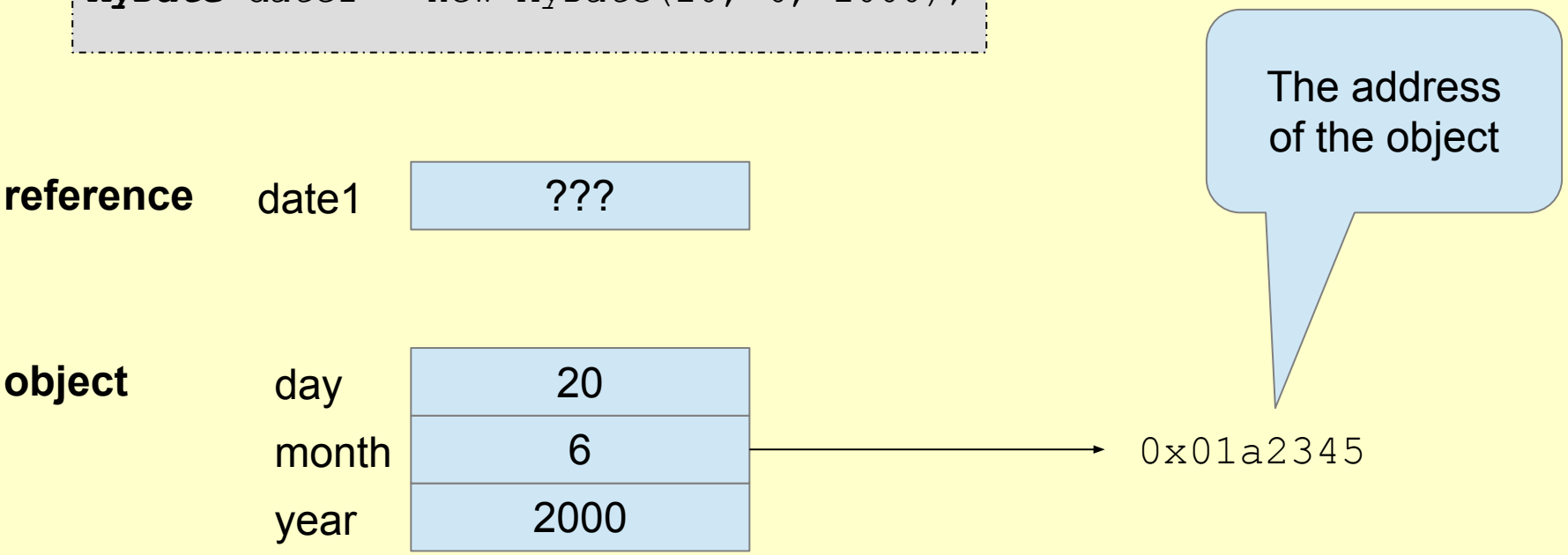
```
MyDate date1 = new MyDate(20, 6, 2000);
```




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

}
```

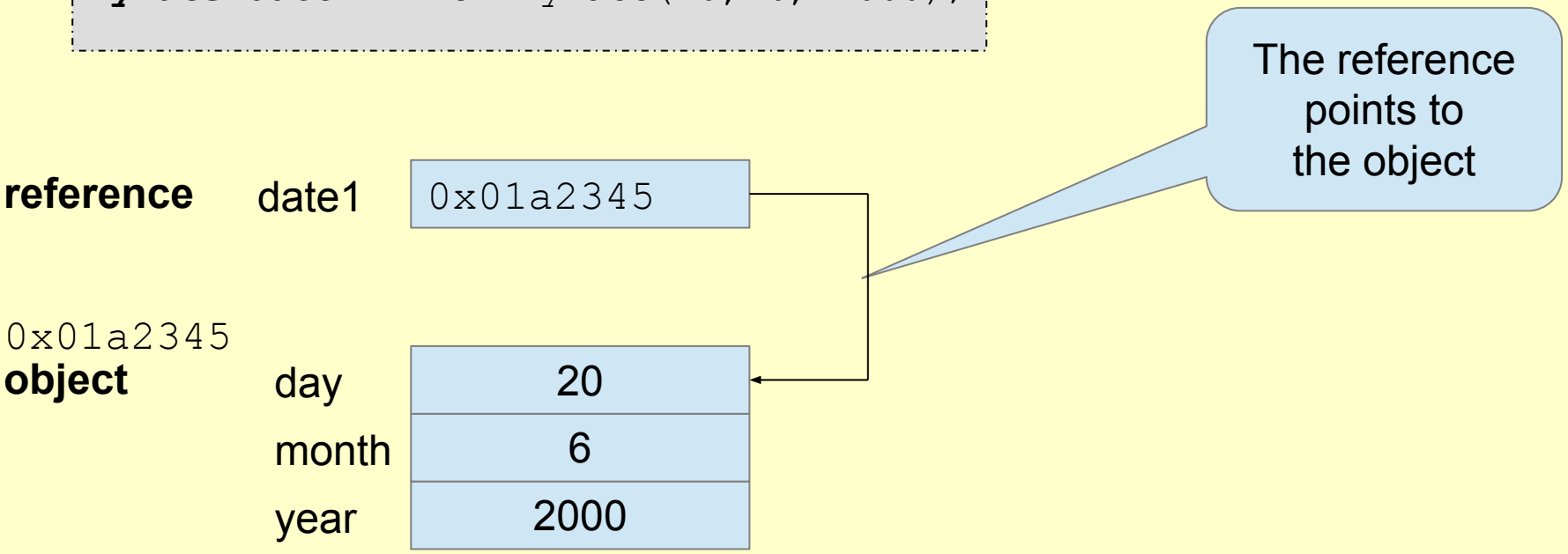
(4) The object reference is returned

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



(5) The reference is assigned to a variable

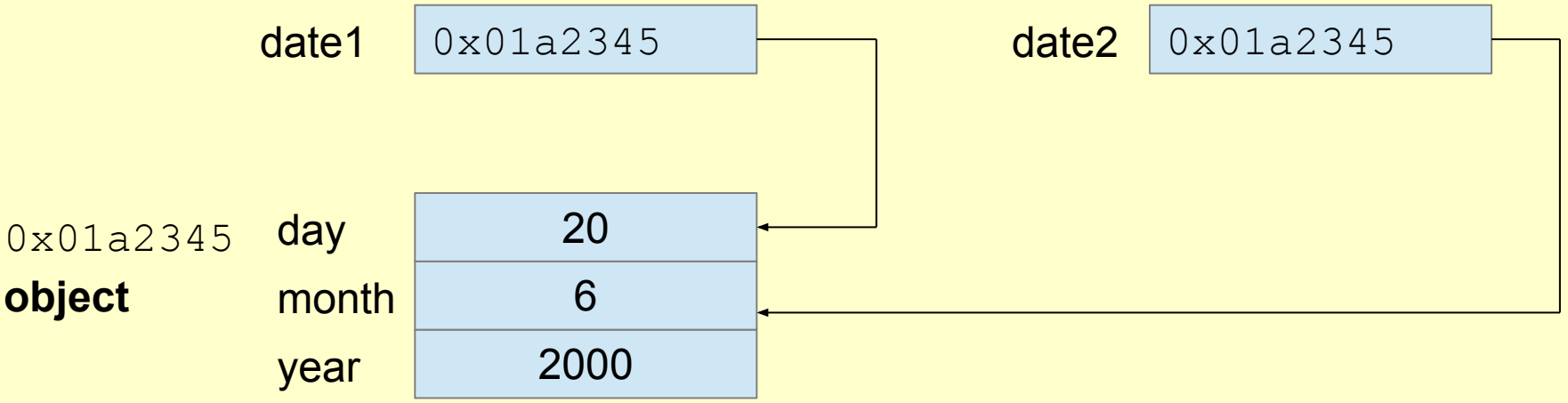
```
MyDate date1 = new MyDate(20, 6, 2000);
```



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

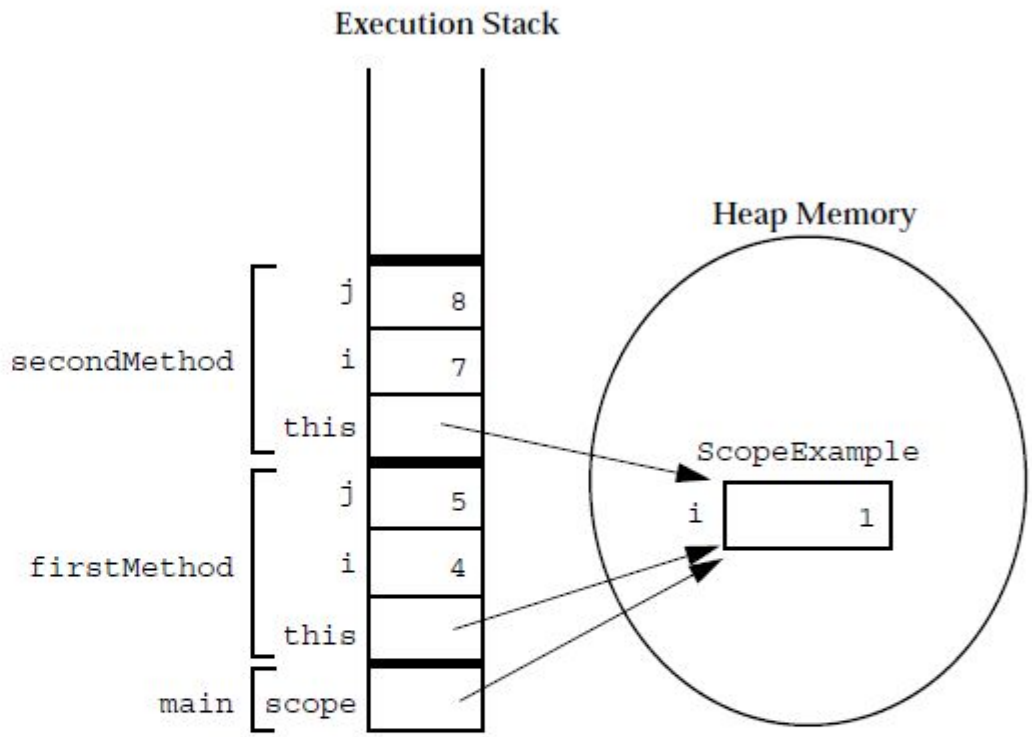
- Packages
  - `ro.sapientia.ms`
- Classes
  - `SavingsAccount`
- Methods
  - `getAmount()`
- Variables
  - `amount`
- Constants
  - `NUM_CLIENTS`

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();
  }
}
```

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

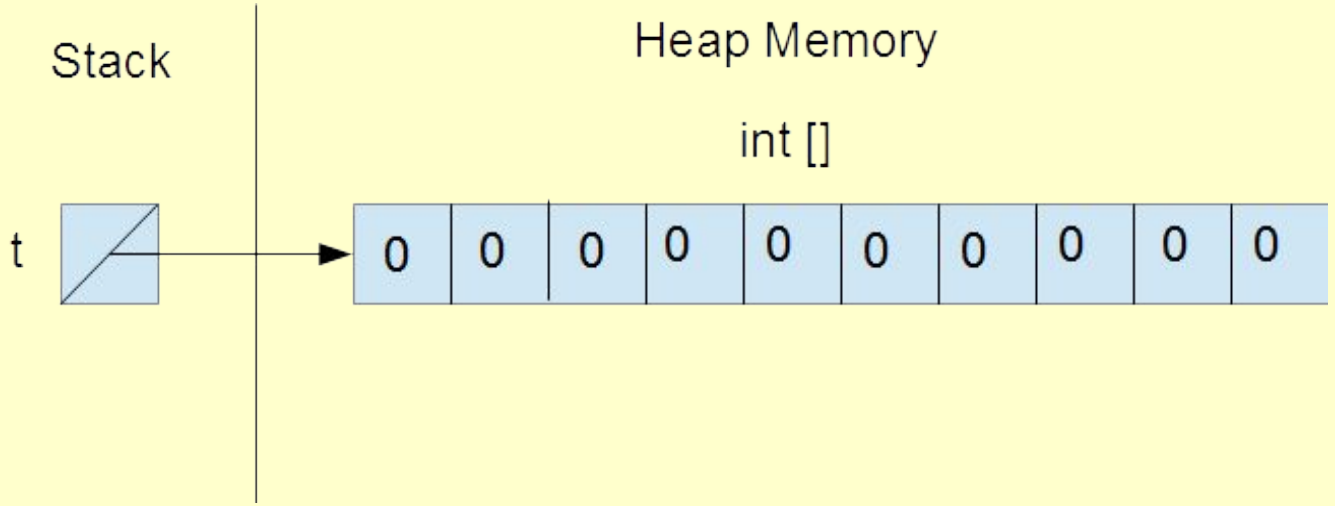
- Arrays are **objects** → are created with **new**
- Example:

```
//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

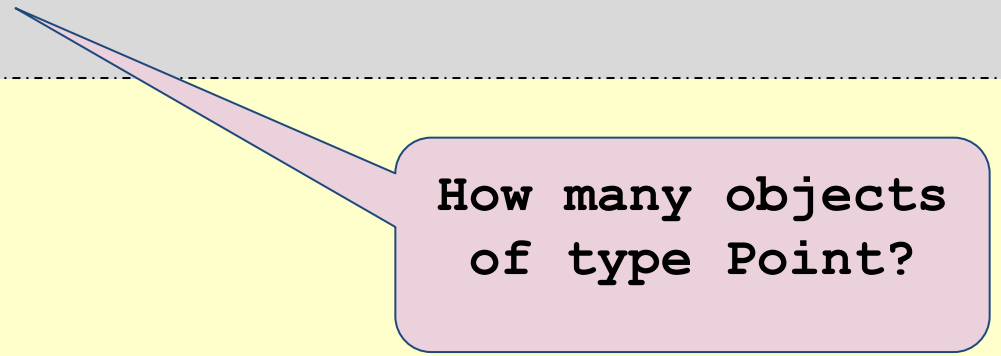
```
//array declaration
Point [] t;

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

How many objects of type Point?

Creating Arrays

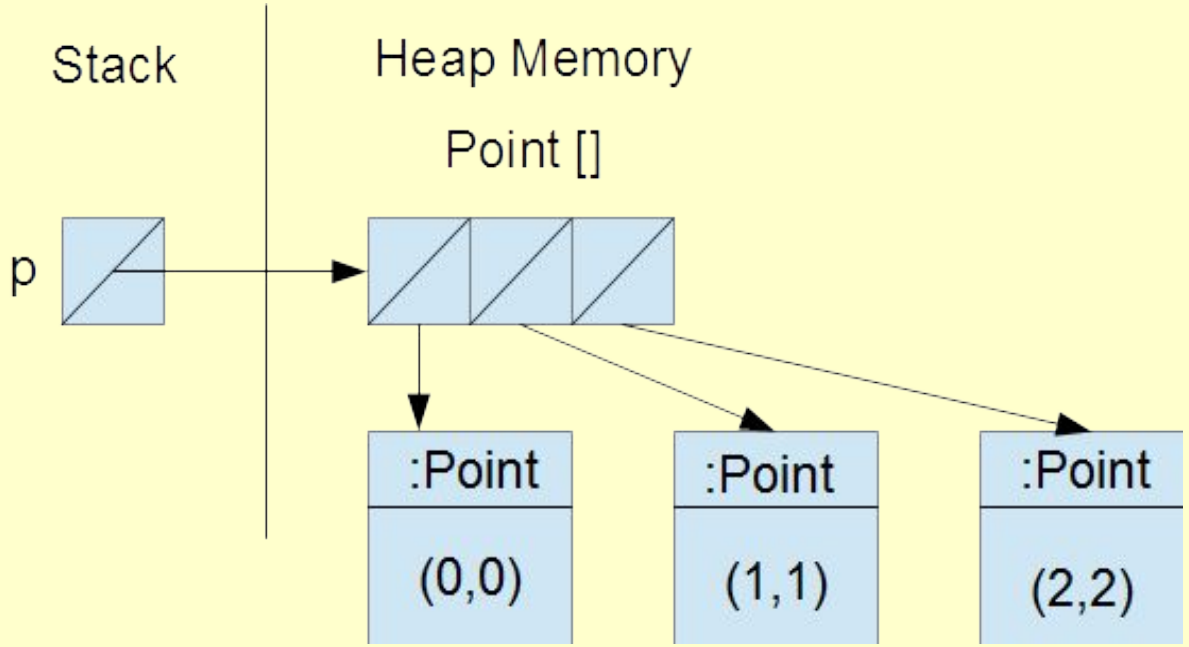
Reference Type

```
//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);
}
```



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

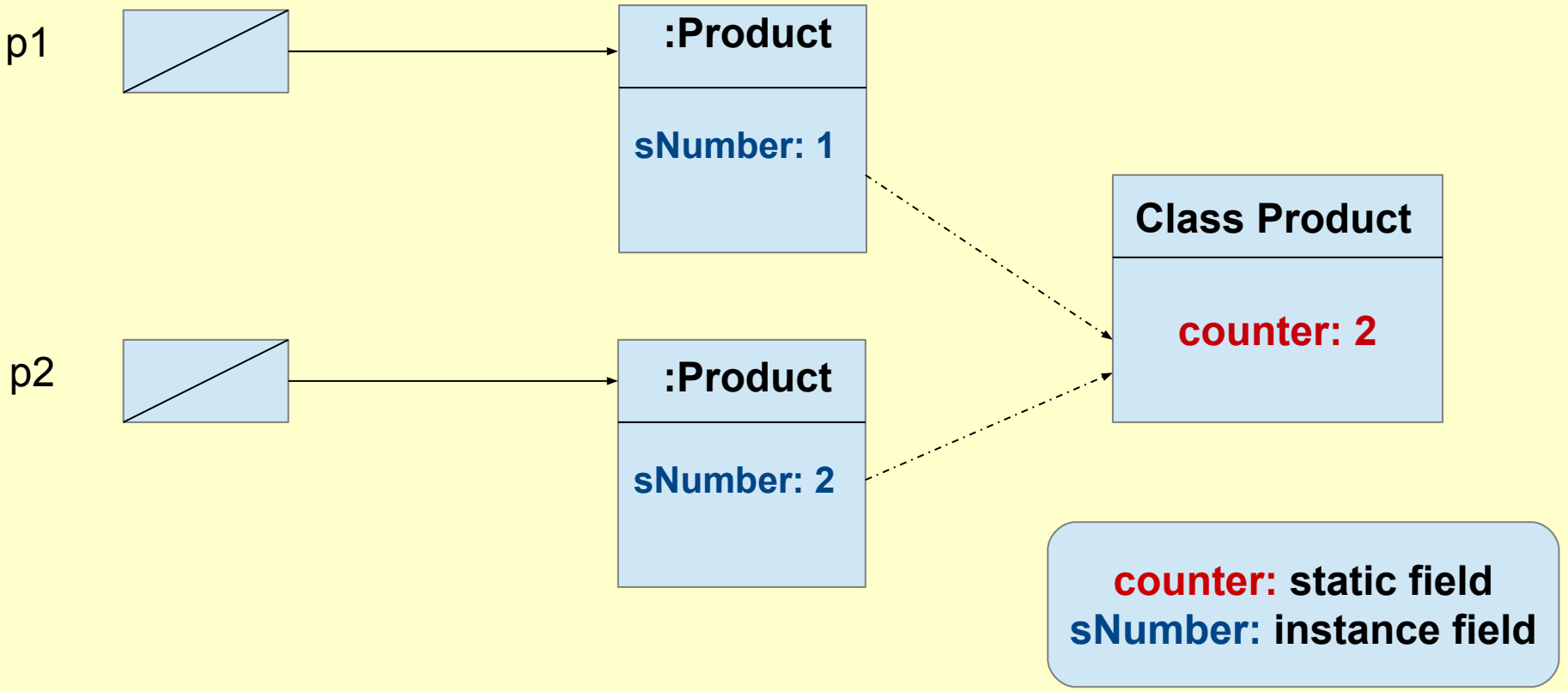
- Create a `Product` class which initializes each new instance with a `serialNumber` `1,2, 3,…`

Solution

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

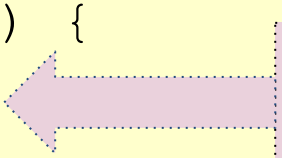
Solution

```
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;
   }
}
```

Output?

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

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

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

```
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;
    }
}
```

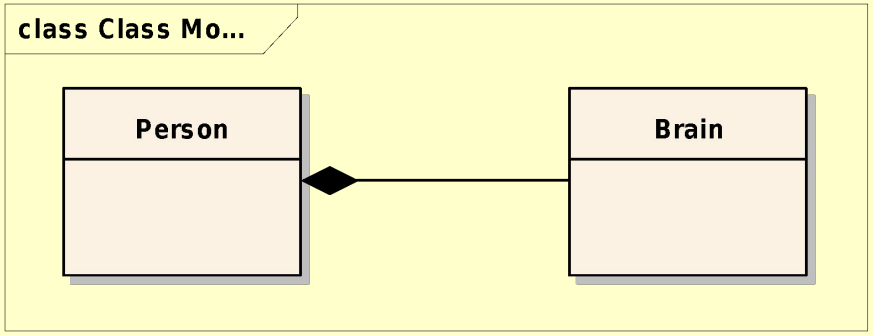
Module 4

Relationships between classes

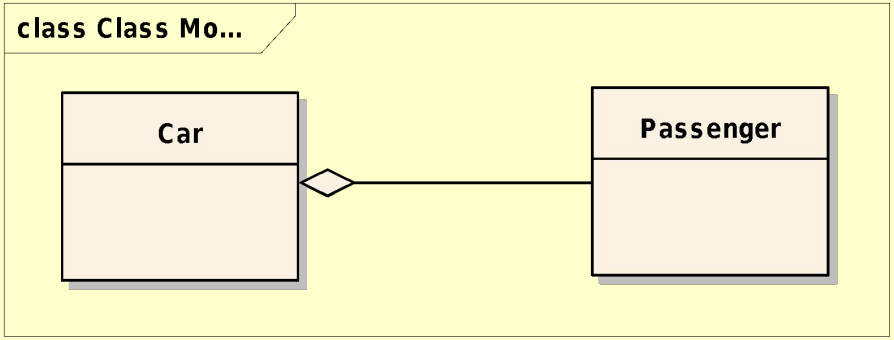
Object-oriented programming

Relationships between classes

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

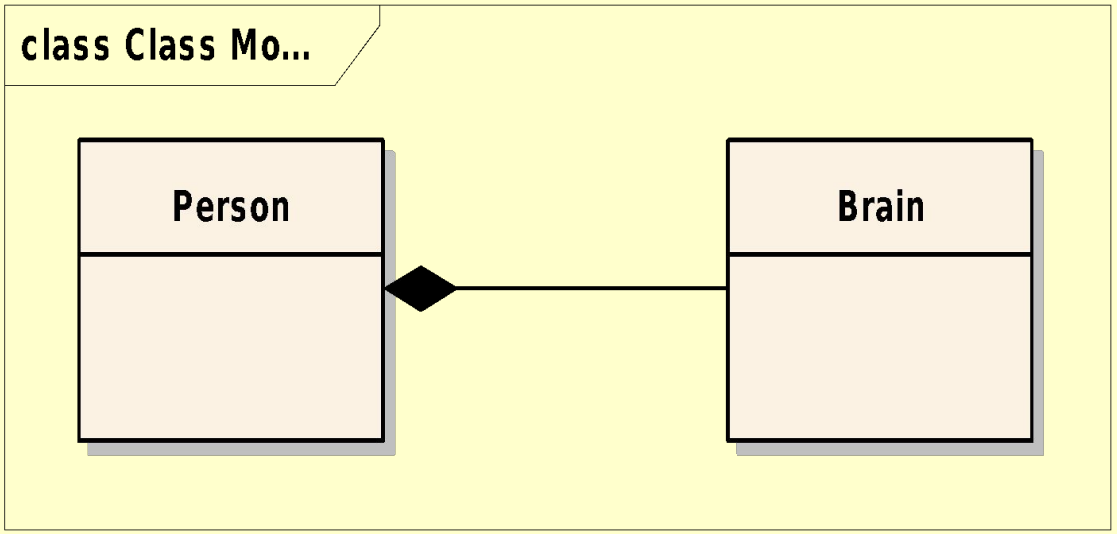


  - Weak – **Aggregation**



Relationships between classes

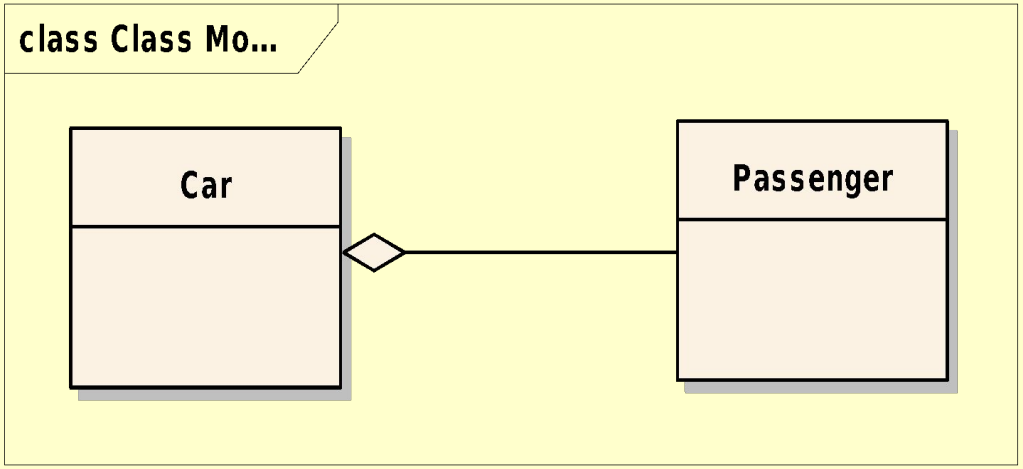
Composition



- Strong type of association
- Full ownership

Relationships between classes

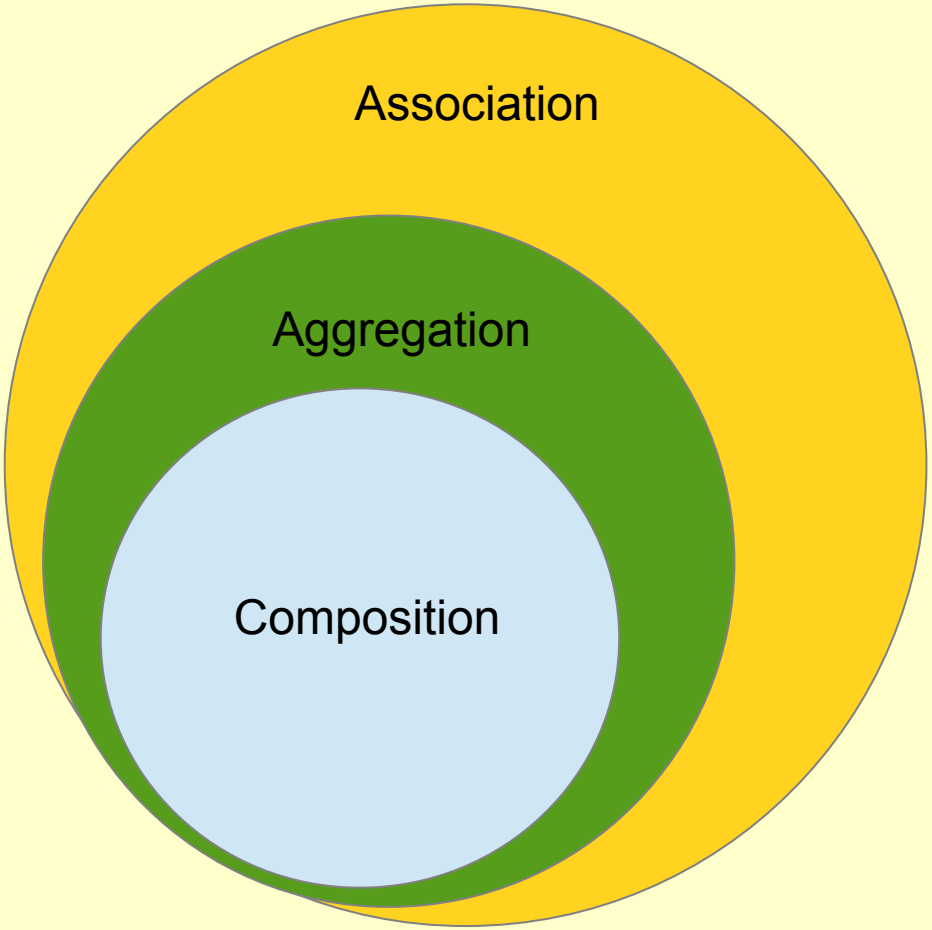
Aggregation



- Weak type of association
- Partial ownership

Relationships between classes

Association – Aggregation - Composition



Relationships between classes

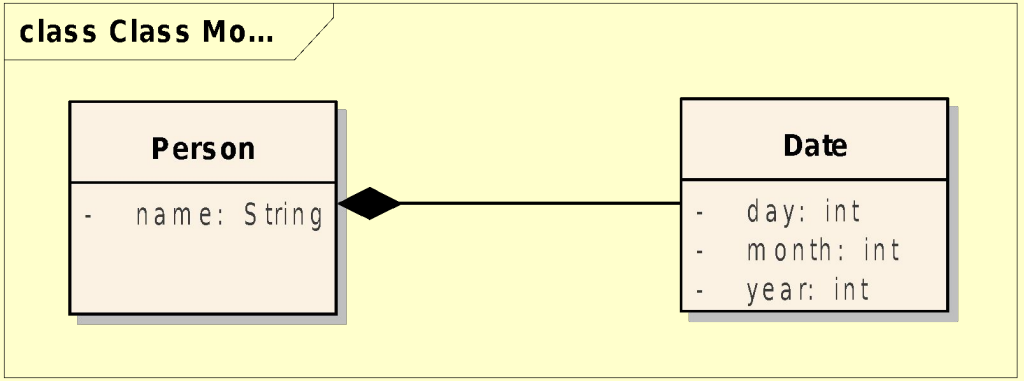
Implementing Associations (1)

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

}
```

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

Relationships between classes

Implementing Associations (2)



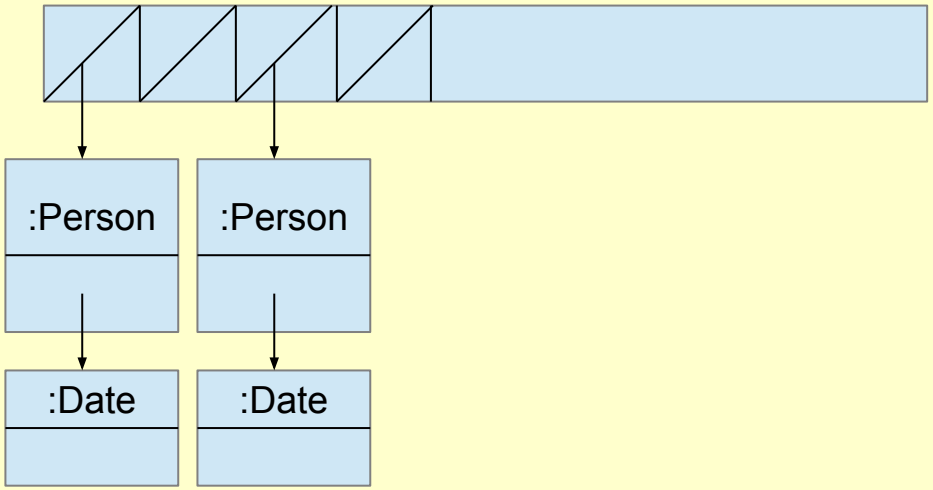

```
public  class Date{
    private int day;
    private int month;
    private int year;
    //...
}
```

```
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
...
```

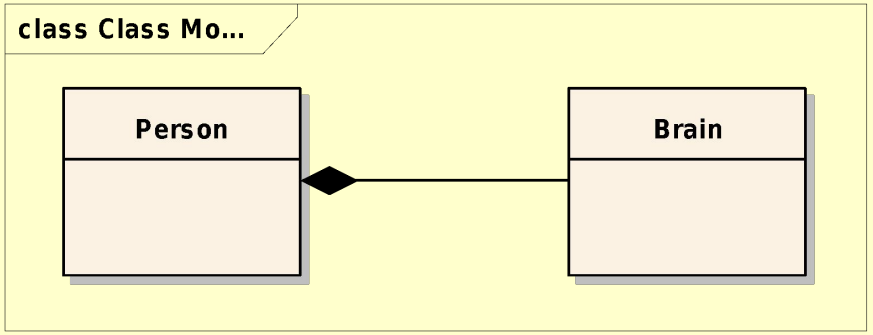


Write a program which reads the data of several persons and constructs an array of Persons.

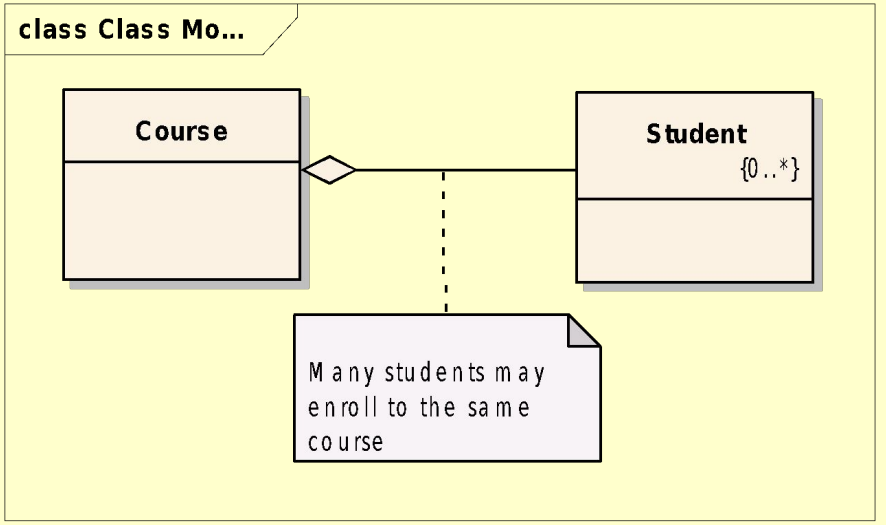
Relationships between classes

Relationship cardinality

- *One-to-one*



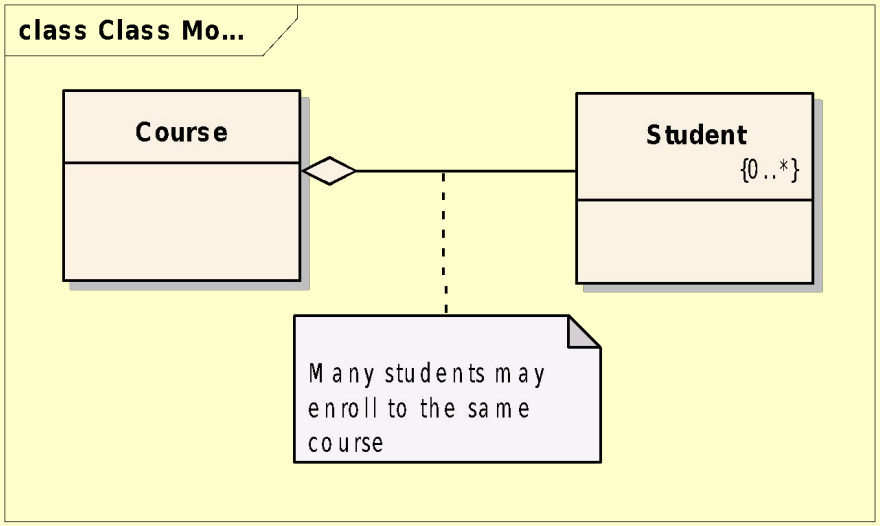
- *One-to-many*



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);
    }


    //...
}
```

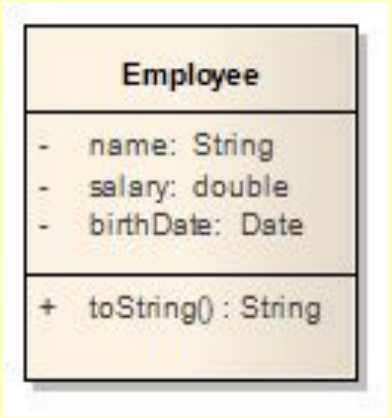
Module 5

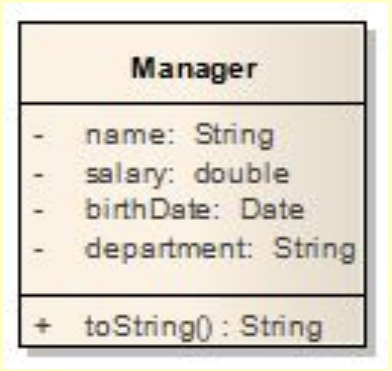
Inheritance, Polymorphism

Outline

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

Problem: *repetition in implementations*

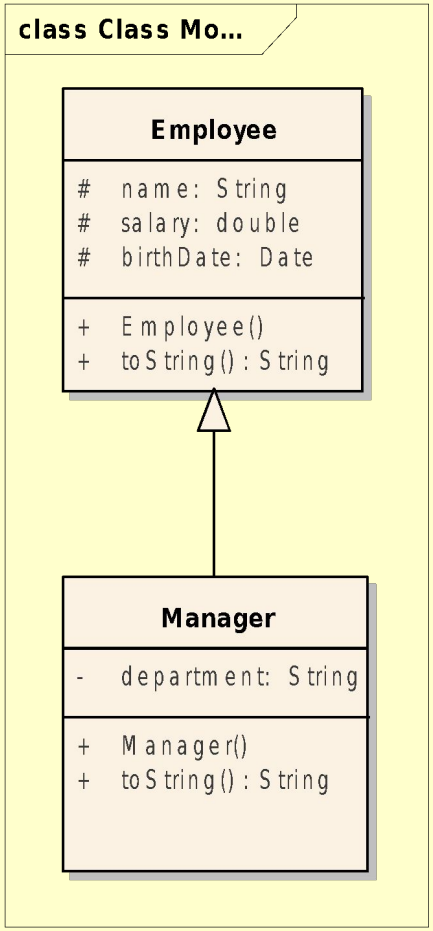





```
public  class Employee{
    private String name;
    private double salary;
    private Date birthDate;

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

```
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;
     }
}
```

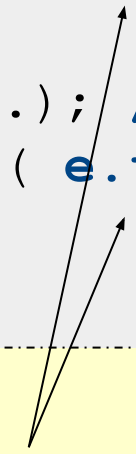
Overridden Methods Cannot Be Less Accessible

```
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?

Polymorhic 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;
   }
}
```

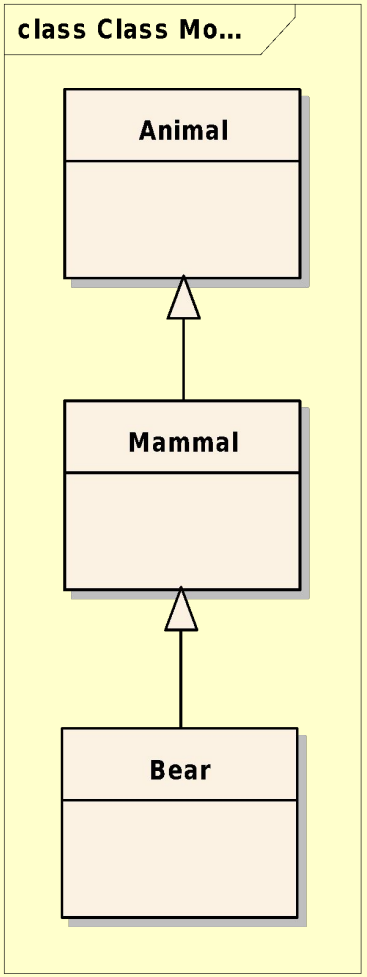
Static vs. Dynamic type of a reference

```
// 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
- Methods **overriding**
  - same name, *same signature*
  - e.g. `toString()`
  - run-time 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

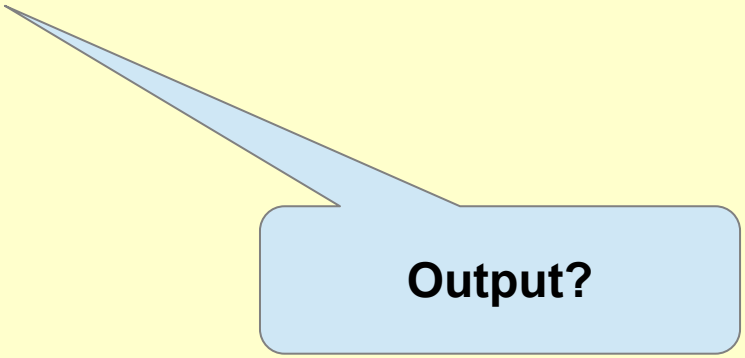
Methods Common to All Objects

- `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));
```



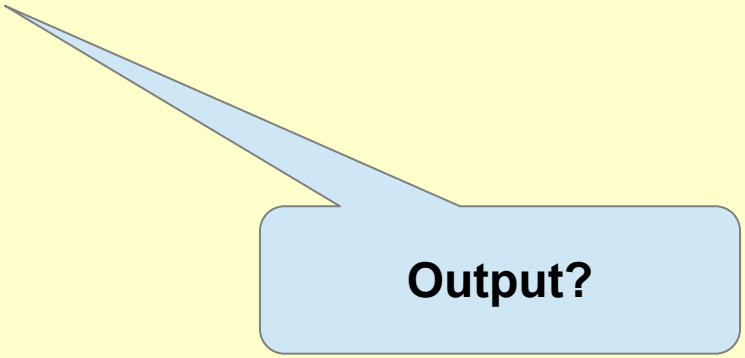
An `equals` example

```
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;
    }

}
```

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));
```



The `equals` method implements an equivalence relation

- **Reflexive**
  - `x.equals(x):true`
- **Symmetric**
  - `x.equals(y):true ↔ y.equals(x):true`
- **Transitive**
  - `x.equals(y):true and y.equals(z):true → 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

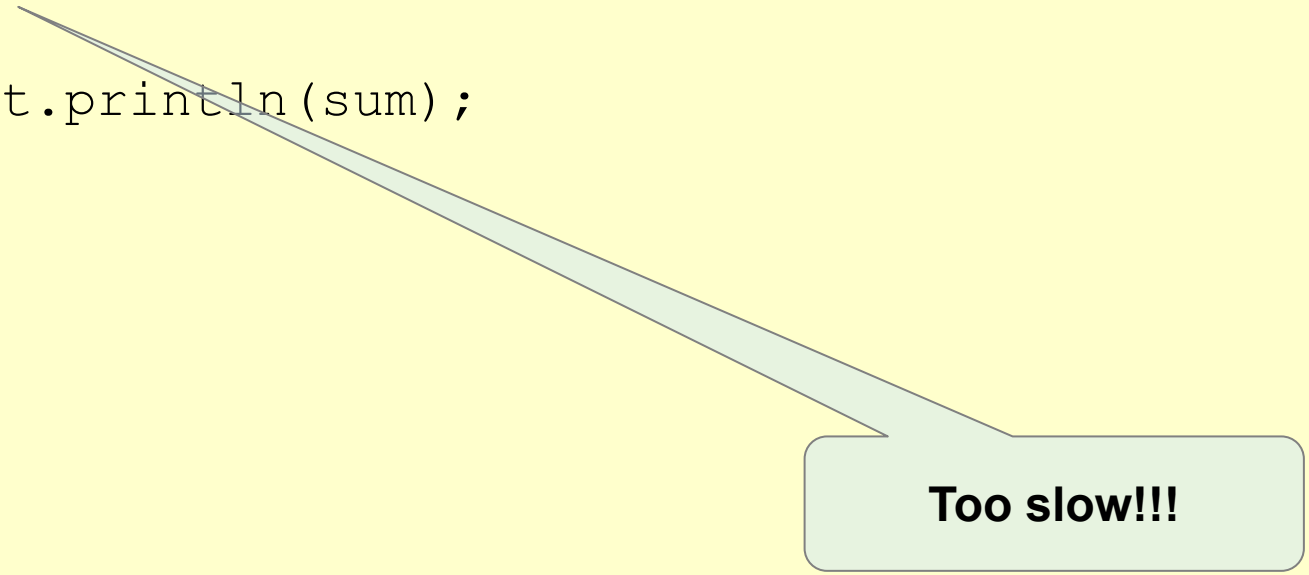
Boxing and Unboxing

```
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);
}
```



Module 6
Interfaces and Abstract Classes

Outline

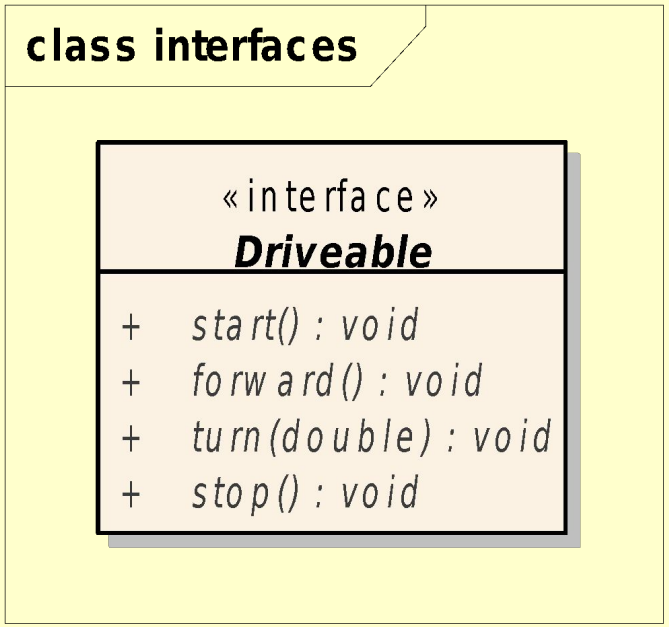
- Interfaces
- Interfaces (since Java 8)
- Abstract classes
- Sorting
  - `Comparable` interface
  - `Comparator` interface

Interfaces

- Properties
  - Define **types**
  - Declare a **set of methods** (*no implementation!*) – ADT – Abstract Data Type
  - Will be **implemented** by classes

The `Driveable` Interface

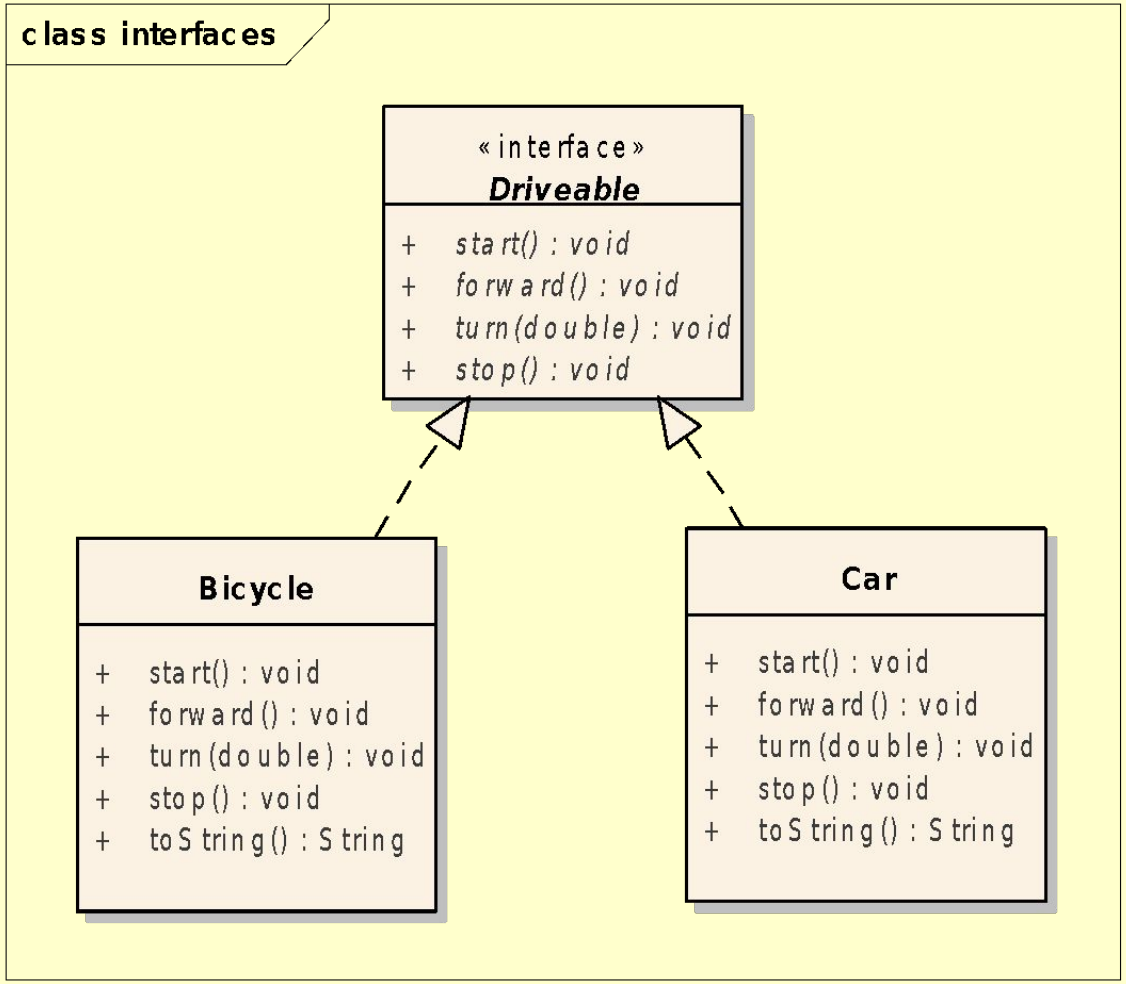
```
public interface Driveable{
      public void start();
      public void forward();
      public void turn( double angle);
      public void stop();
 }
```



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;`

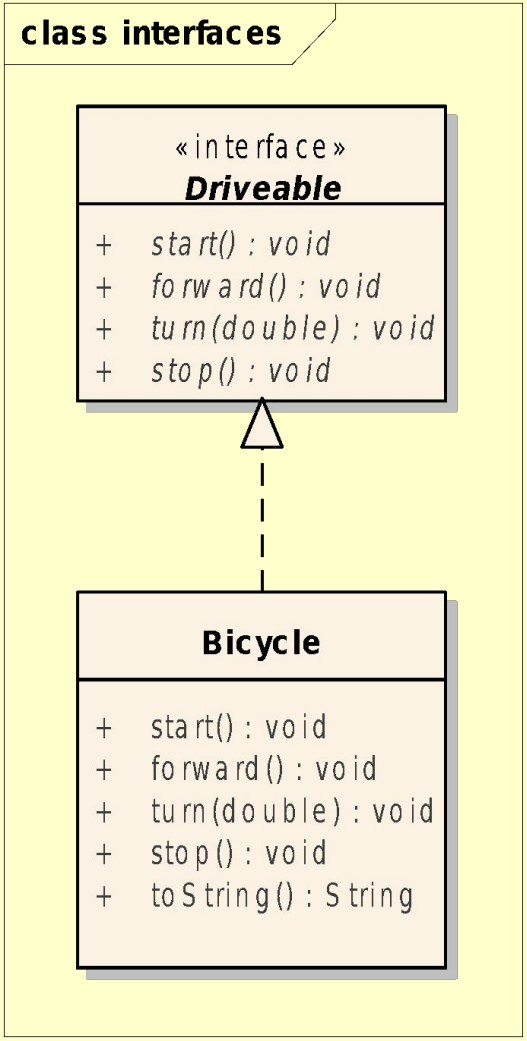
b) `Driveable a = new Driveable();`

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



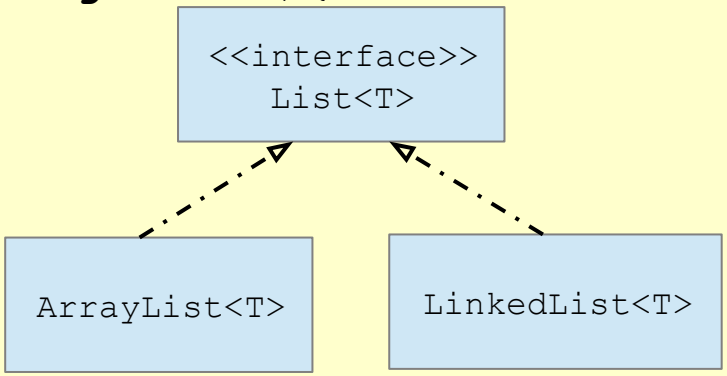
Polymorphic Argument

```
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 );
      }
   }
}
```




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

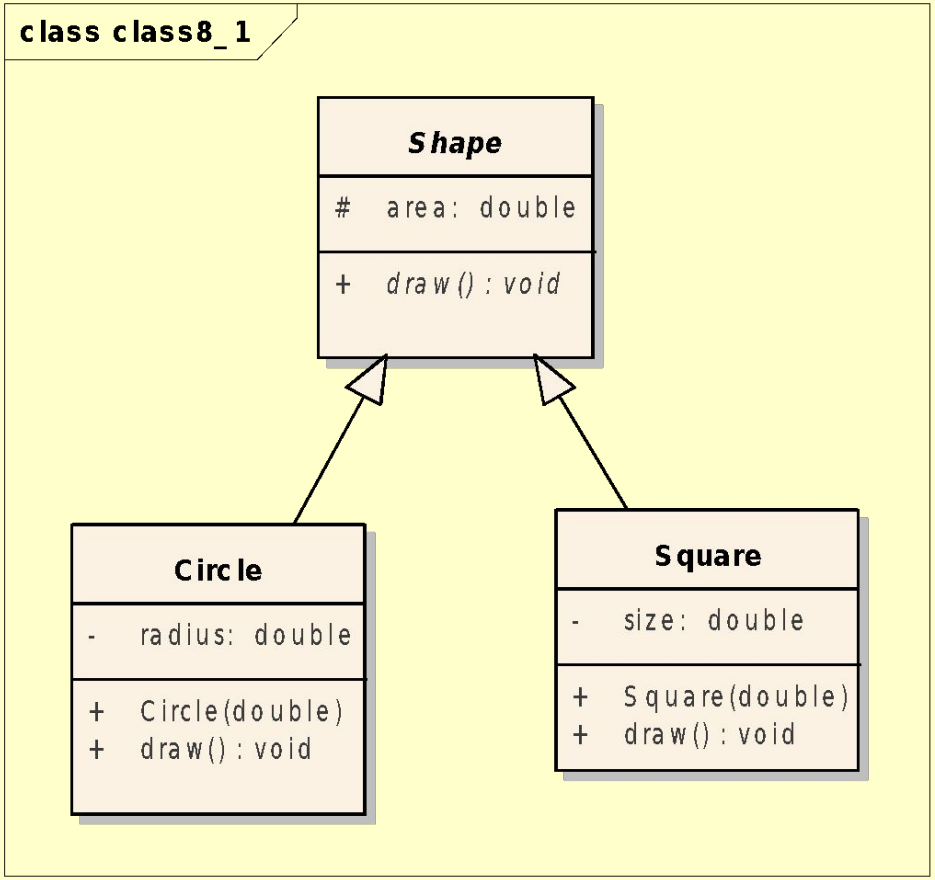
```
public interface MatrixOperations{

    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



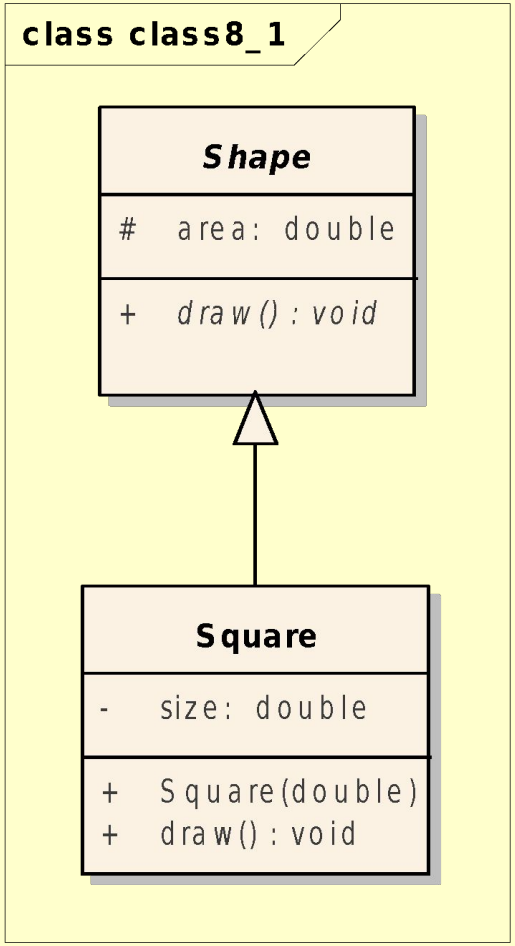
Abstract Classes

```
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	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;

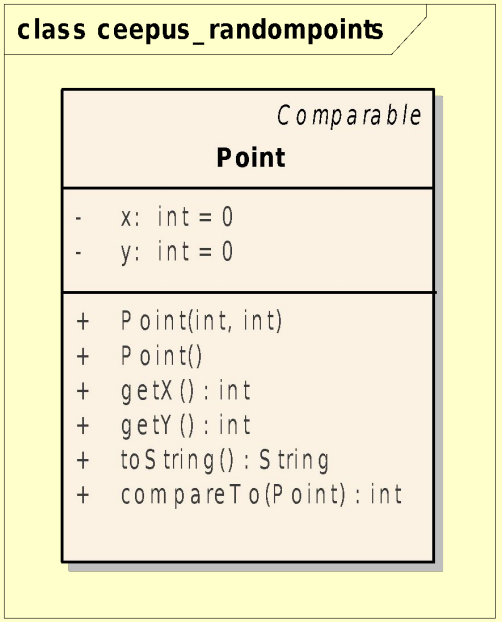
The `Comparable<T>` interface

```
interface Comparable<T> {

   int compareTo(T o);
}
```

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

The Comparable<T> interface

```
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

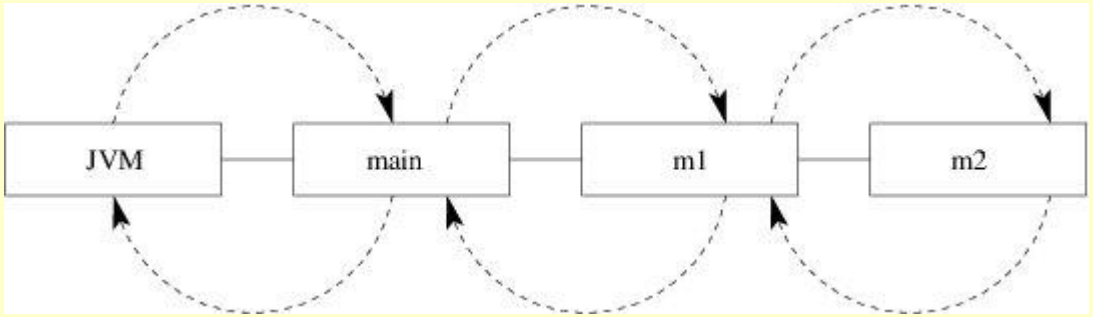
```
public class AddArguments3 {
   public static void main(String[] args) {
         int sum = 0;
         for( String arg: args ){
         try{
             sum += Integer.parseInt( arg );
             } catch( NumberFormatException e ){
                 System.err.println(arg+"is not an integer");
             }
         }
         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 (1)

```
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 `finally` clause (2)

```
static String readFirstLineFromFile(String path)
                                                throws IOException {
    BufferedReader br = new BufferedReader(
                                              new FileReader(path));
    try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

Closing resources

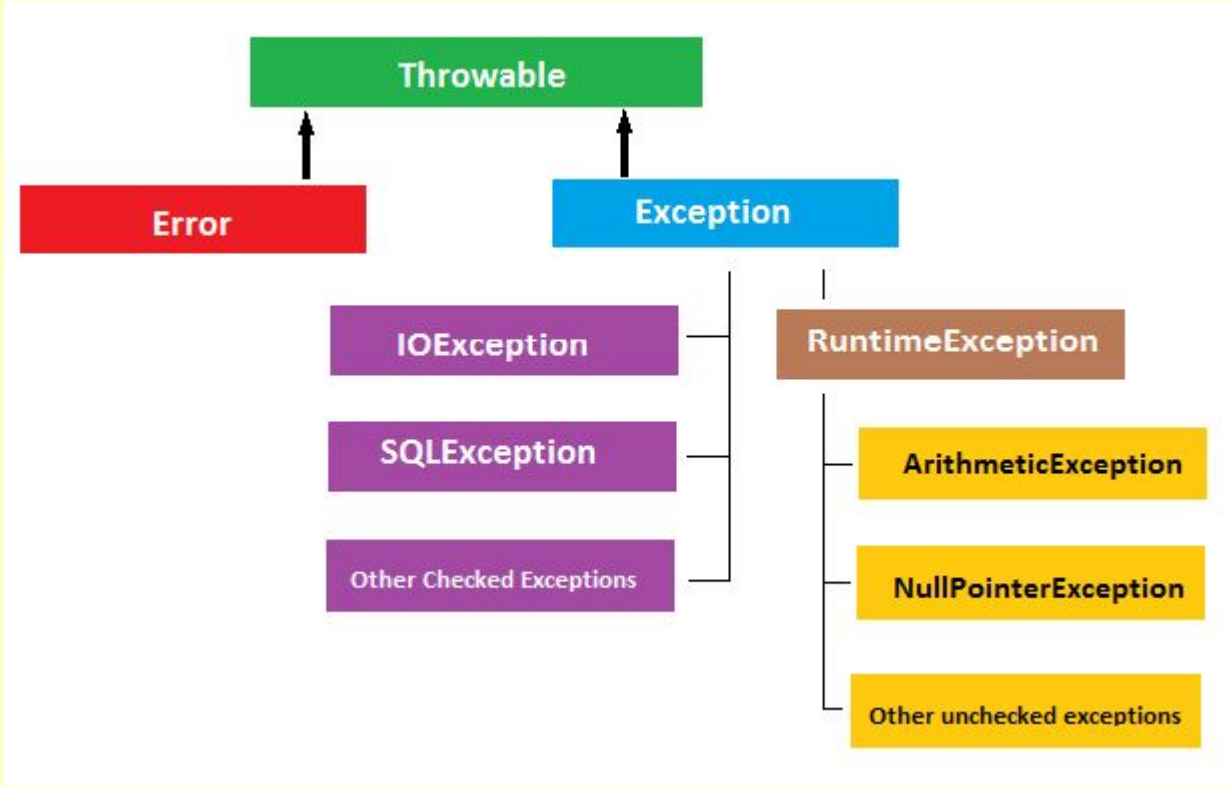
The try-with-resources Statement

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

```
static String readFirstLineFromFile(String path)
                                        throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

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!!!

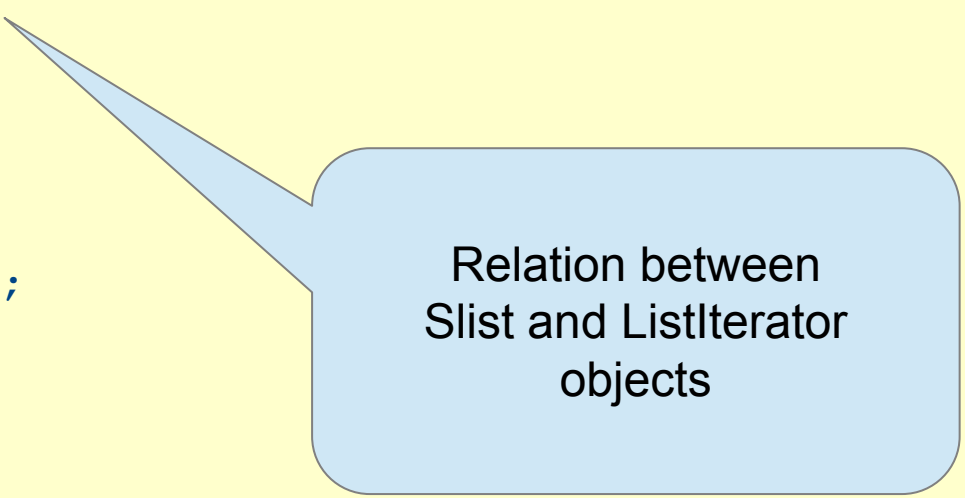
The Iterator interface

```
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;
     }
  }
}
```



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

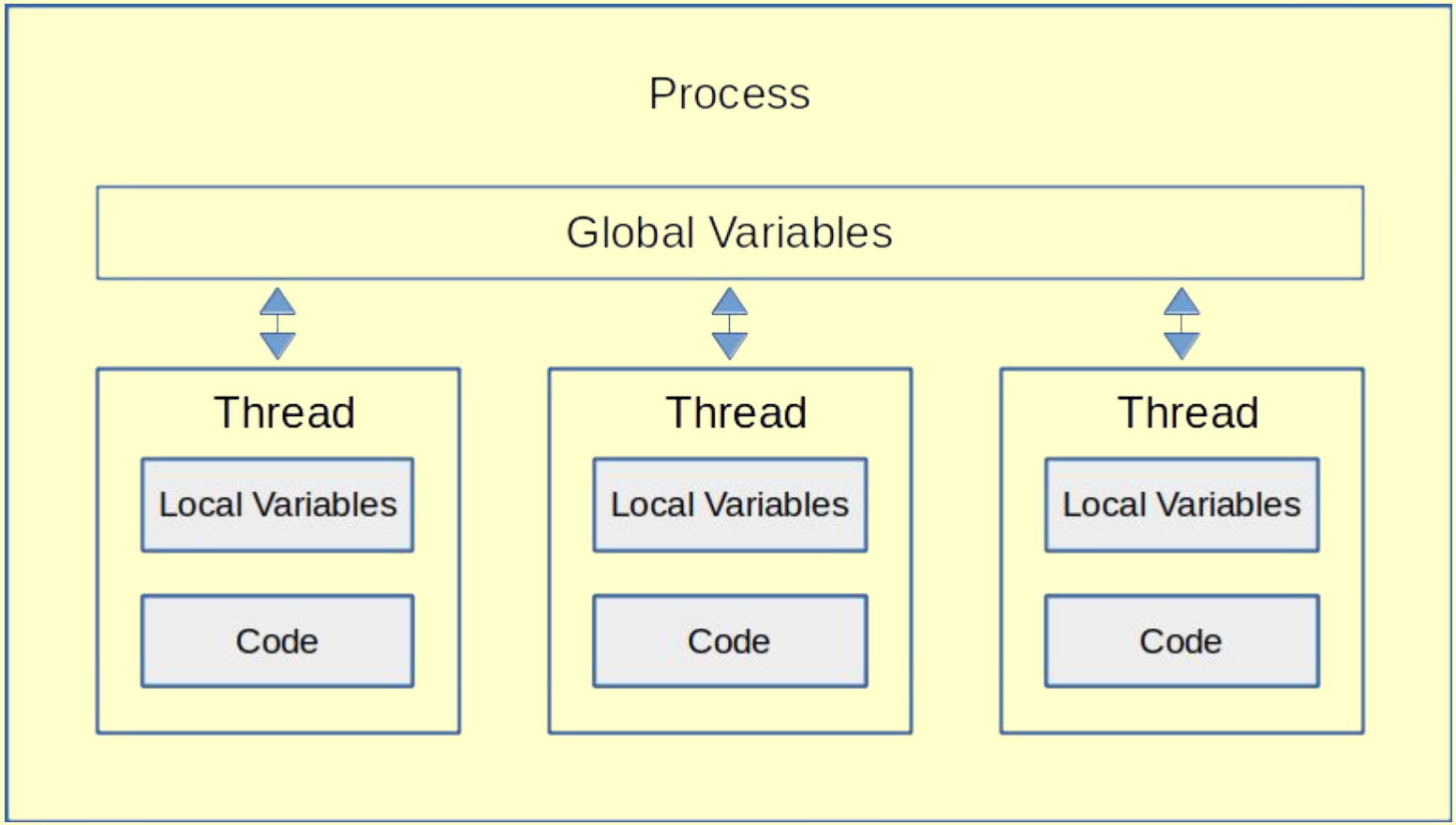
Outline

- **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?



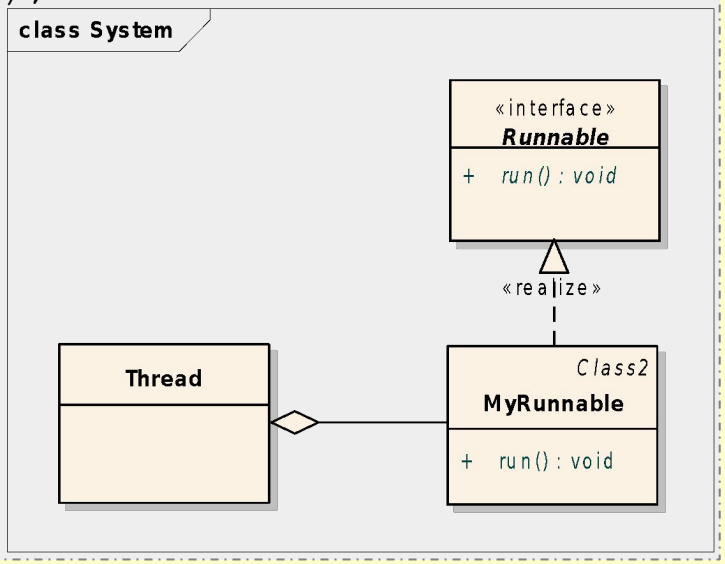
Threads

`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();
   }
}
```

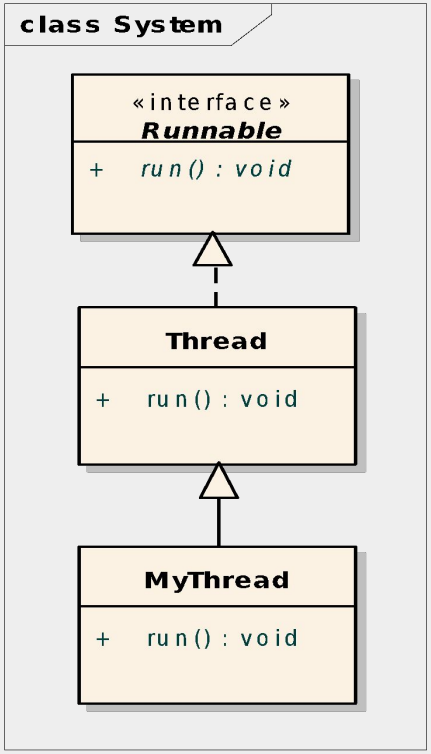
Output?

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()

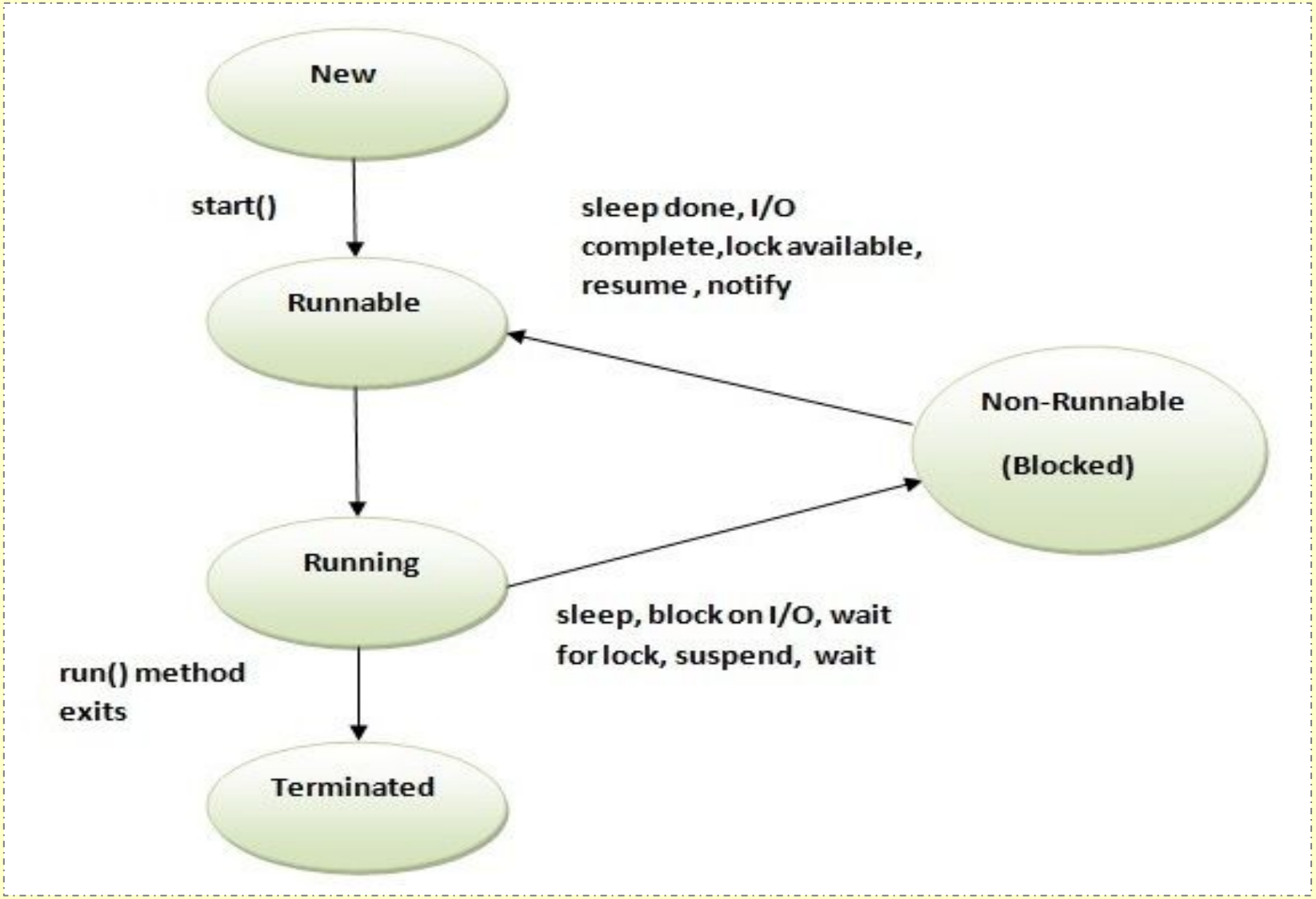
```
private static class ForeverRunnable implements Runnable {
  public void run() {
    while (true) {
       System.out.println(Thread.currentThread().getName() +
                          ": " + System.currentTimeMillis());
       try {
         Thread.sleep(5000);
       } catch (InterruptedException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

interrupt()

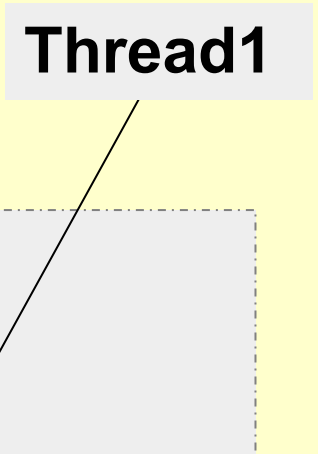
```
private static class ForeverRunnable implements Runnable {
  public void run() {
    while (true) {
       System.out.println(Thread.currentThread().getName() +
                          ": " + System.currentTimeMillis());
       try {
             Thread.sleep(5000);
       } catch (InterruptedException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

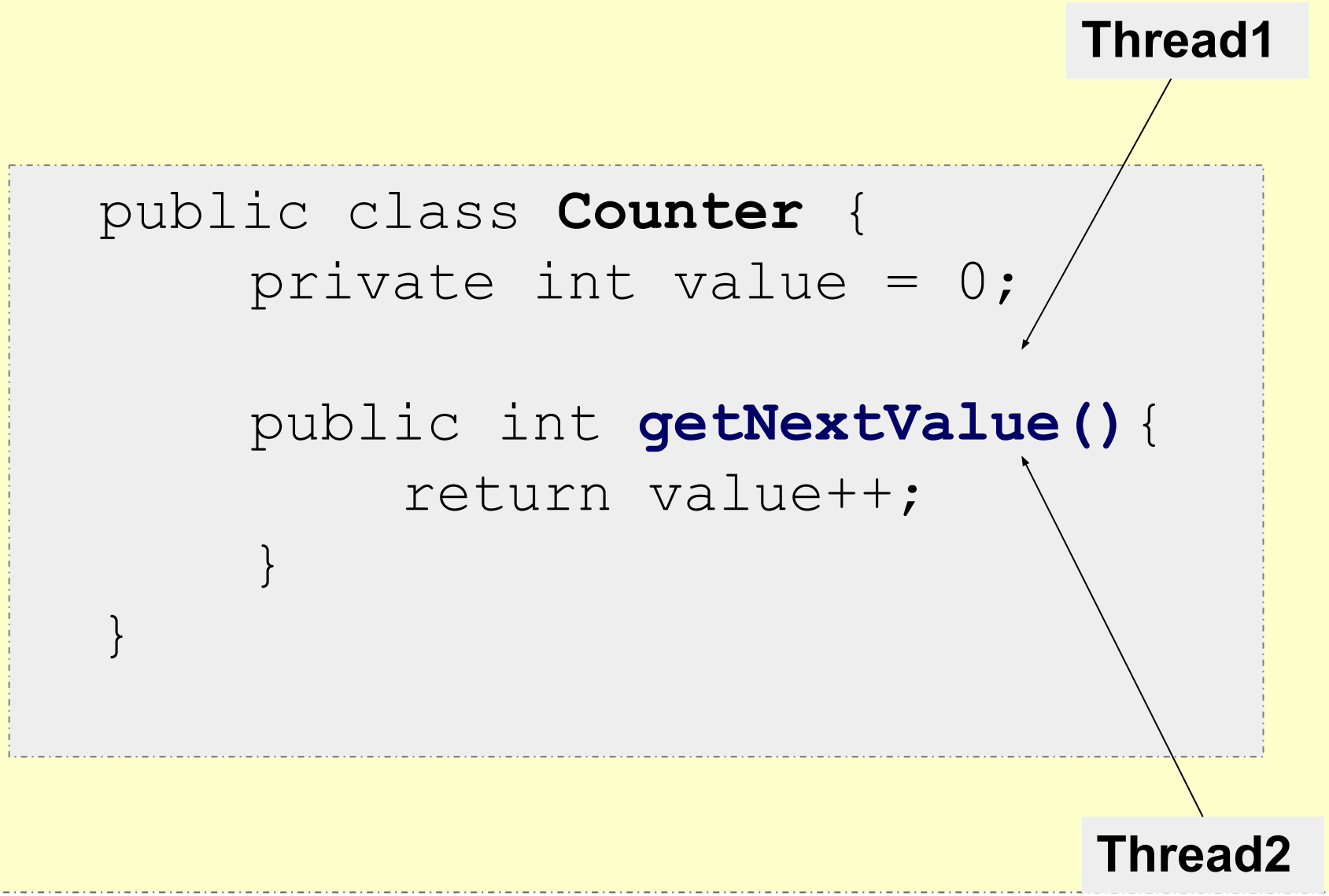


Need for synchronization




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

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

Need for synchronization



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`

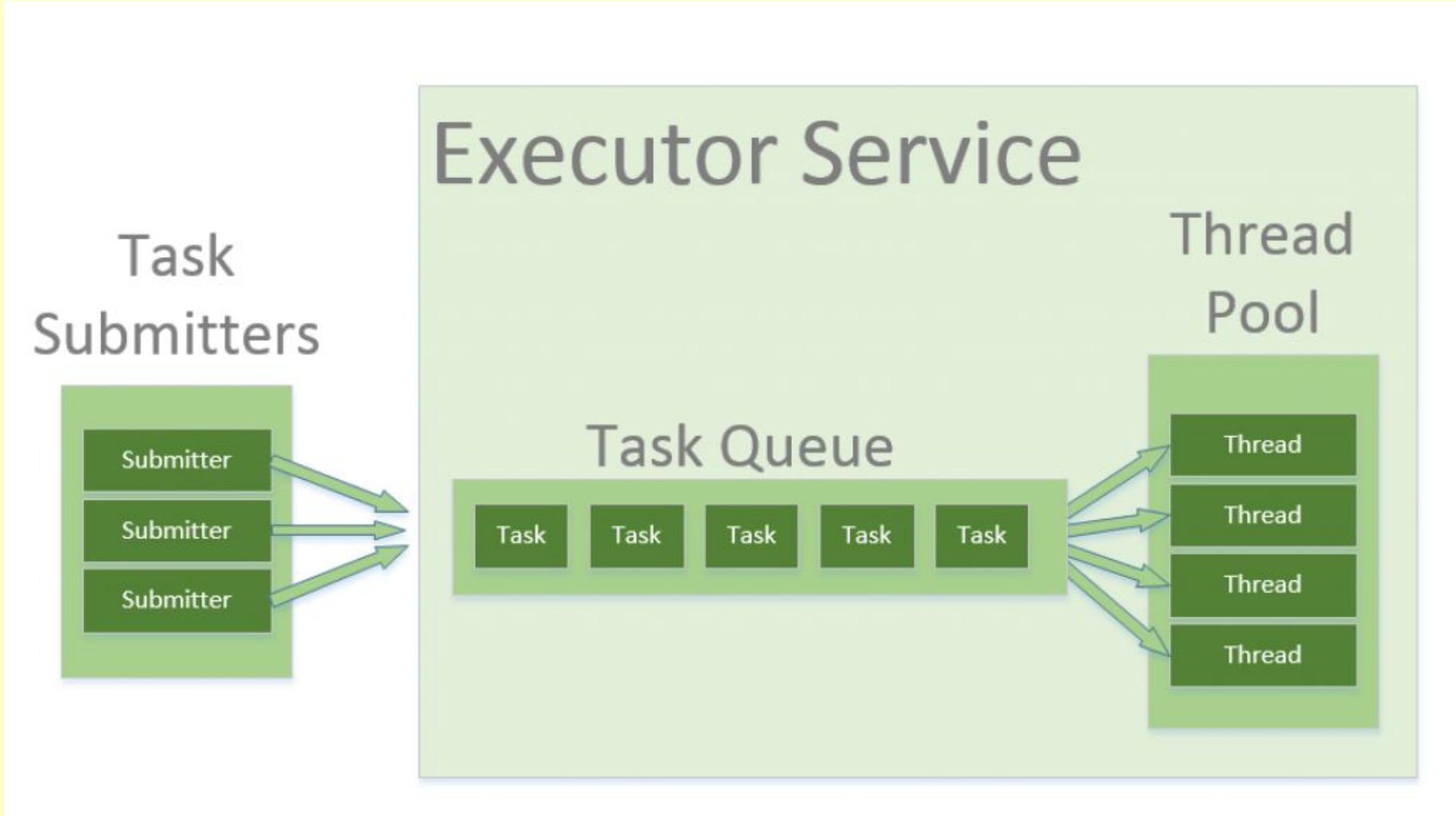
Thread pool

- 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

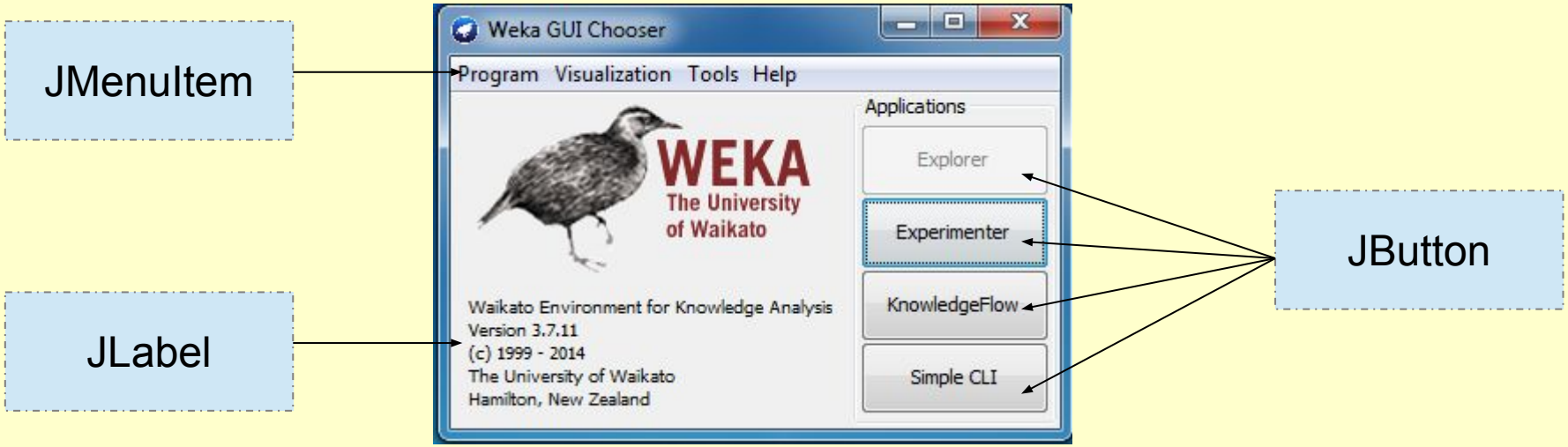
Swing

Outline

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

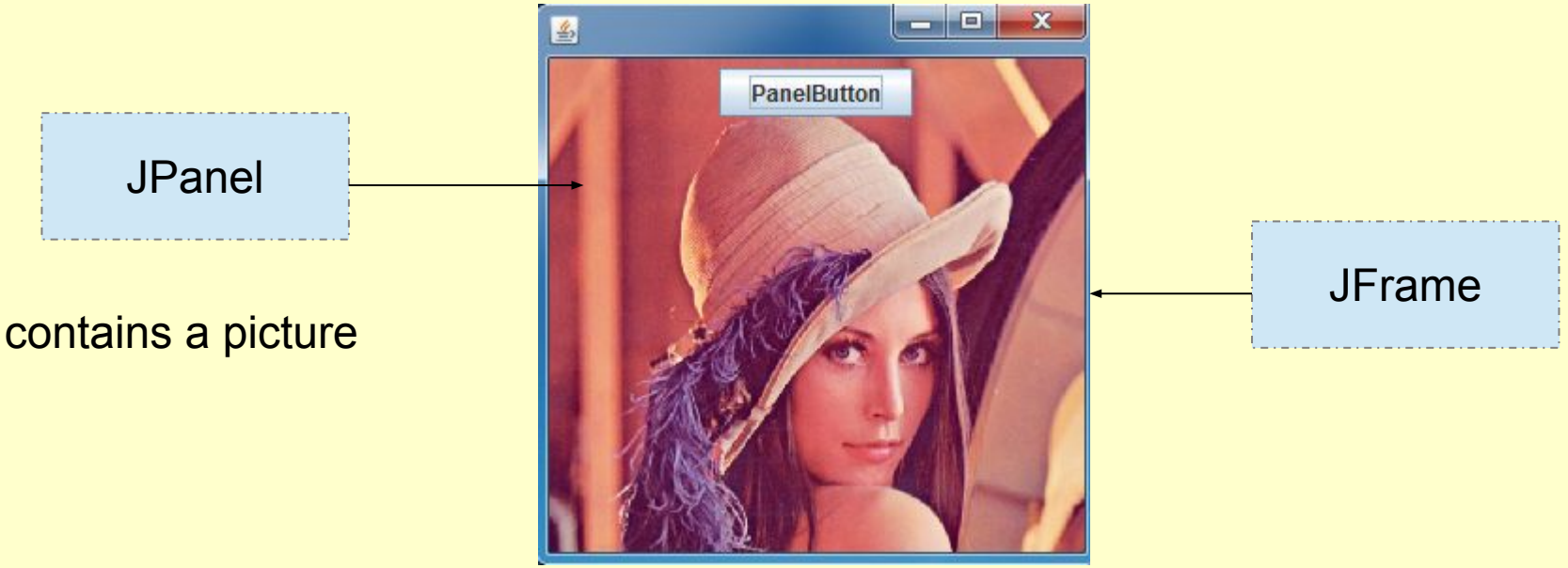
Component

- 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



The first GUI program

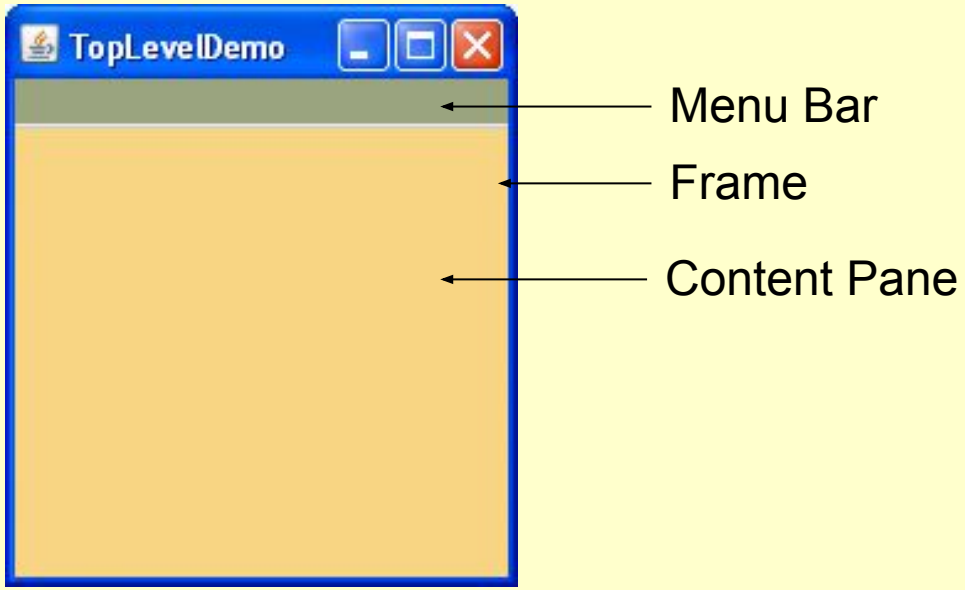
```
public static void main(String[] args) {
        JFrame f = new JFrame("The First Swing
                                        Application");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setBounds( 100,100, 300, 300);
        f.setVisible(true);
  }
```

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)

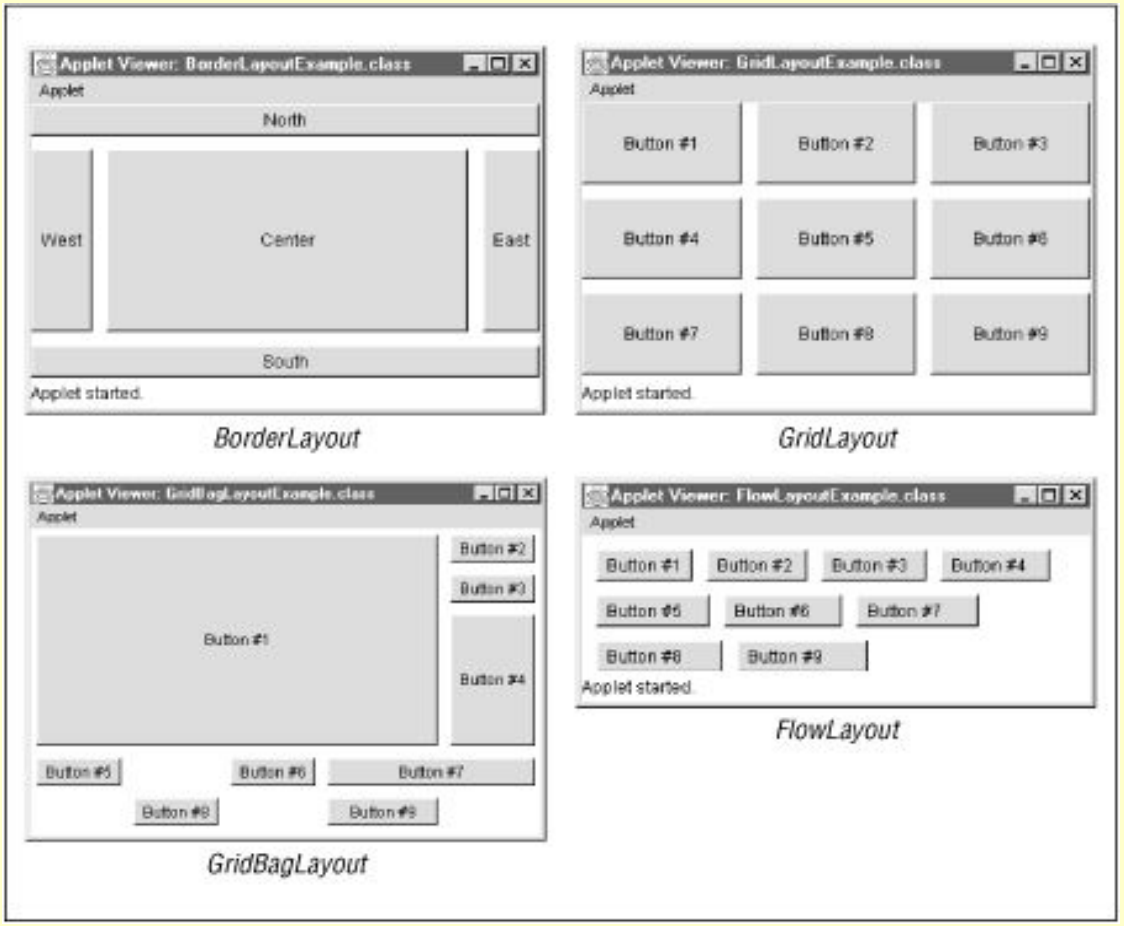
```
JFrame f = new JFrame("The First Swing Application");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

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

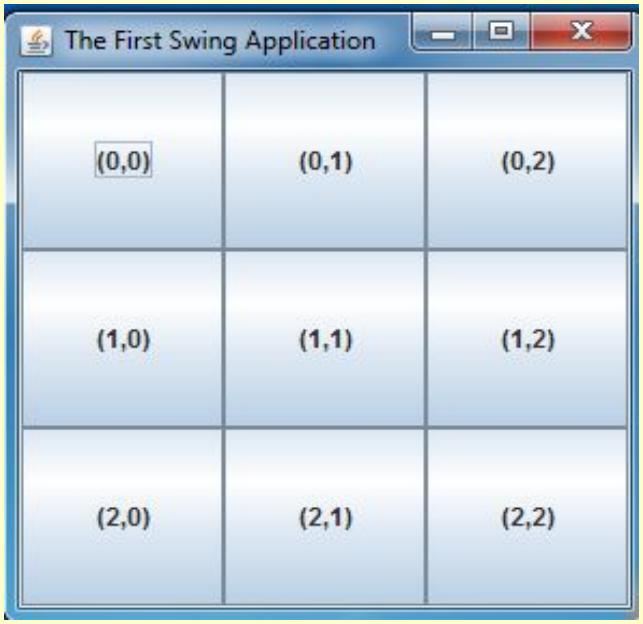
- FlowLayout
- BorderLayout
- GridLayout
- GridBagLayout



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
  - FrameAggregation
- Inheritance
  - FrameInheritance

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();
    }
}
```

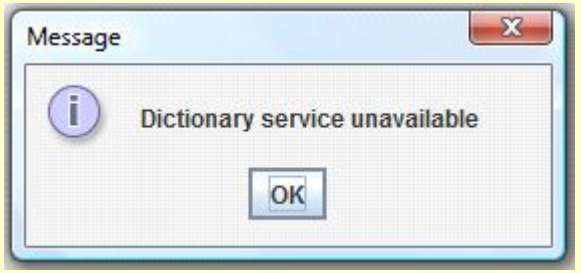
Menus

```
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)

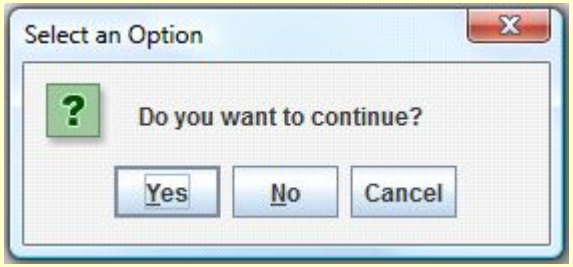
```
JOptionPane.showMessageDialog(
    Component parent, String message);
```



Dialogs

JOptionPane (2)

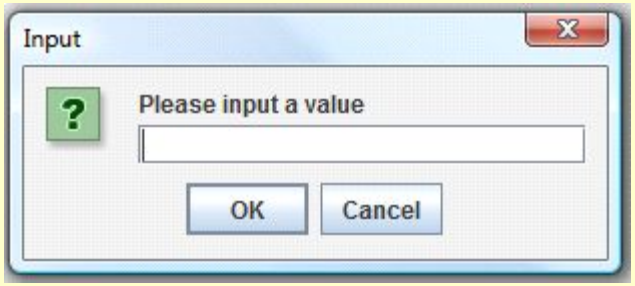
```
int result =
JOptionPane.showConfirmDialog(
          Component parent, String message);
Result:
    YES_OPTION (0), NO_OPTION (1), CANCEL_OPTION (2)
```



Dialogs

JOptionPane (3)

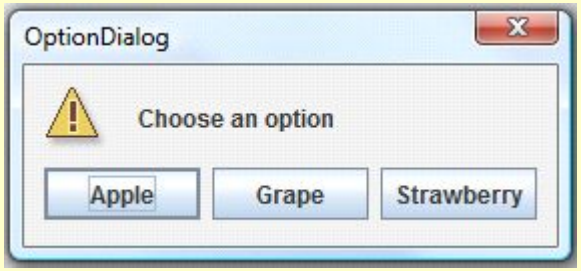
```
String value=
    JOptionPane.showInputDialog("Please input a value");
```



Dialogs

JOptionPane (4)

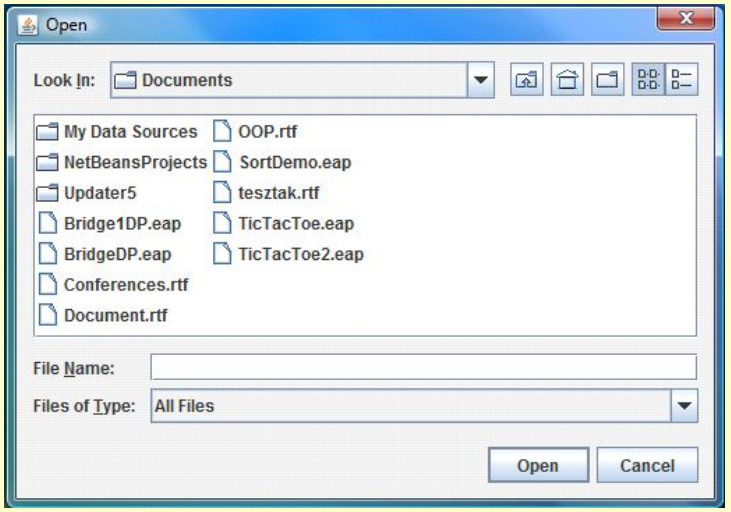
```
String options[]={"Apple", "Grape", "Strawberry"};

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



Dialogs

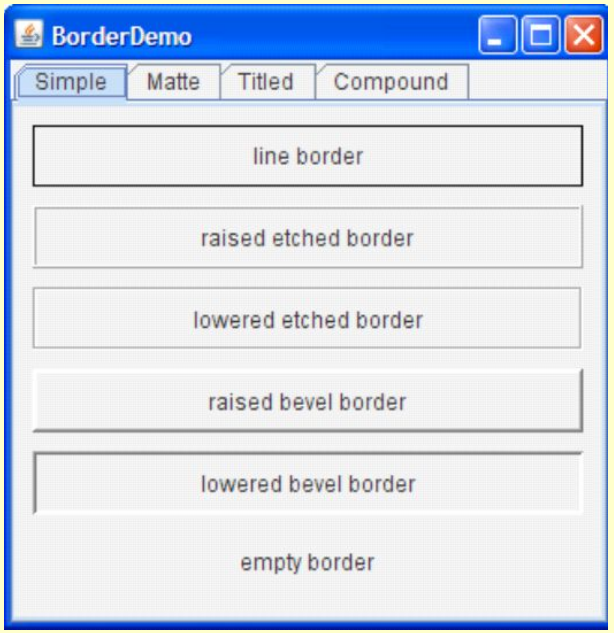
Chooser

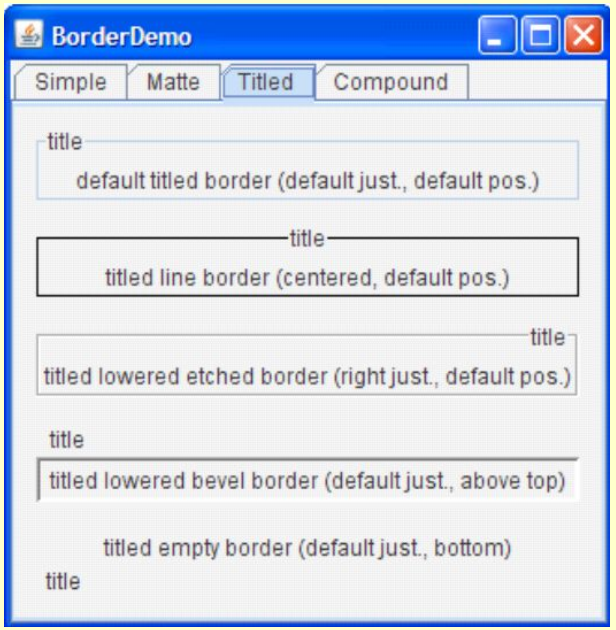
```
JFileChooser chooser = new JFileChooser();
int returnVal = chooser.showOpenDialog(parent);
if(returnVal == JFileChooser.APPROVE_OPTION) {
       System.out.println(
    "You chose to open this file: " +
          chooser.getSelectedFile().getName());
}
```



Borders

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





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

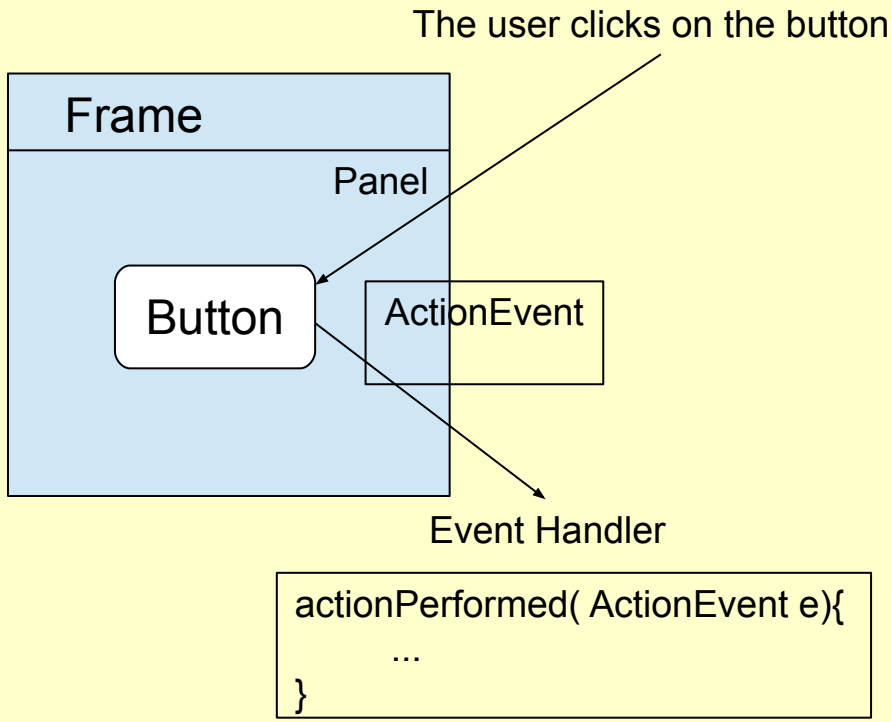
Custom properties

- (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");
```

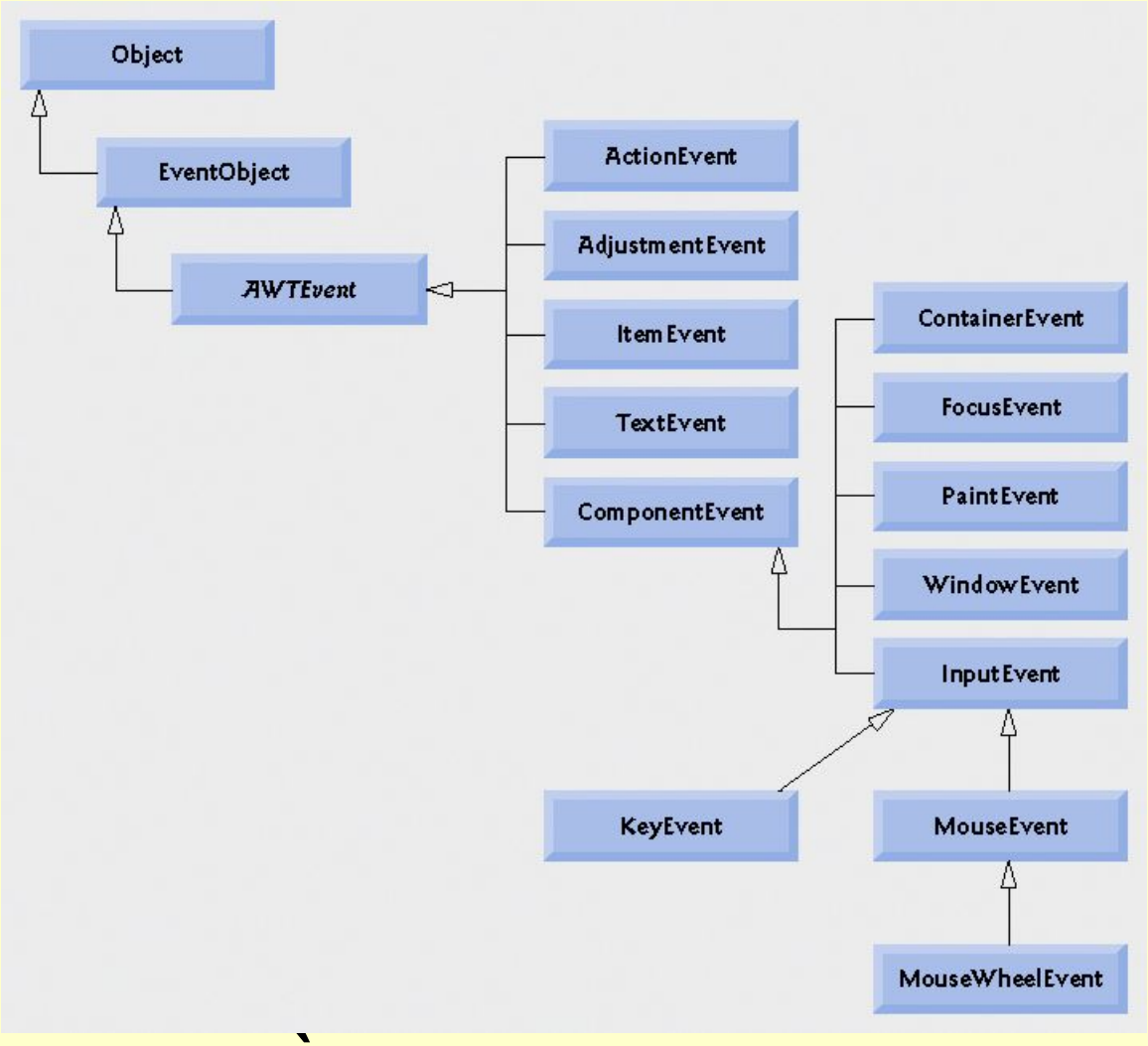
Event Handling

- **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



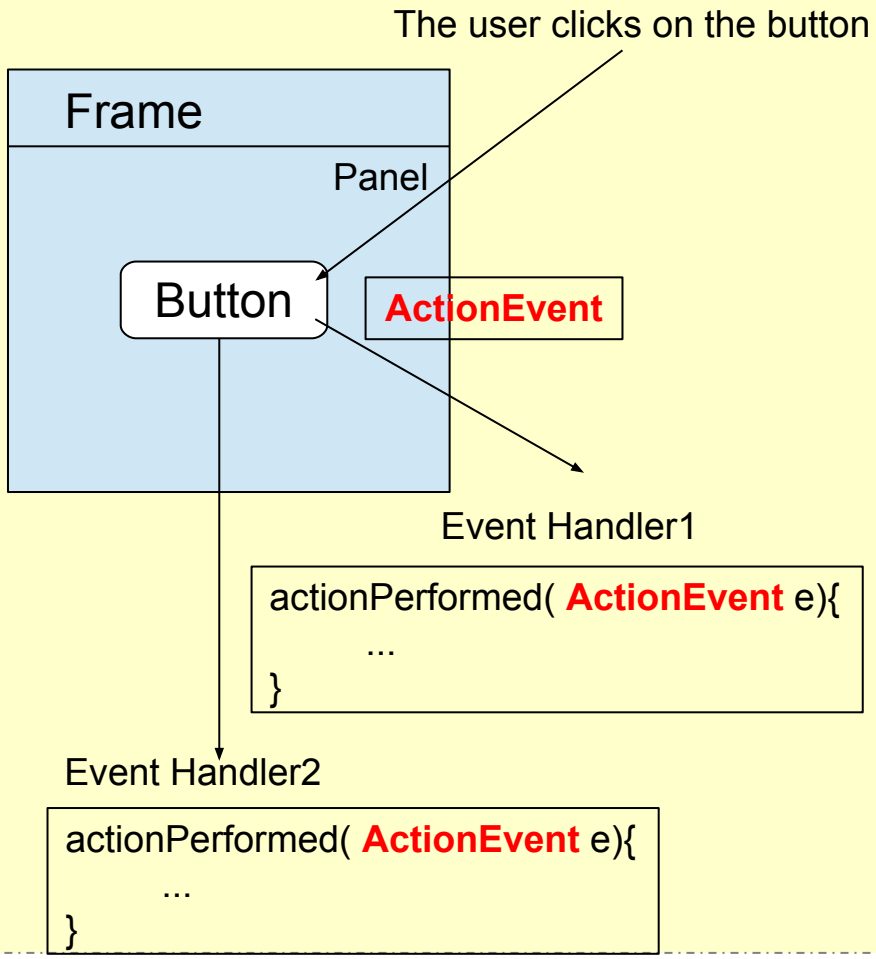
Event Types

- 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

```
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 handler

Event source

(I) Definition of an **anonymous inner class** which implements ActionListener interface

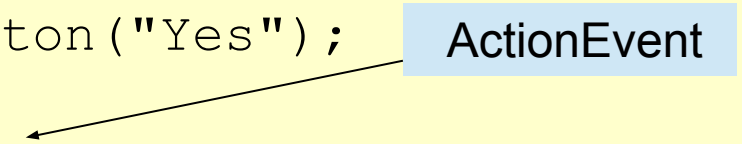
(II) Creation of an instance from that anonymous inner class

(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");
 }
);
```



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

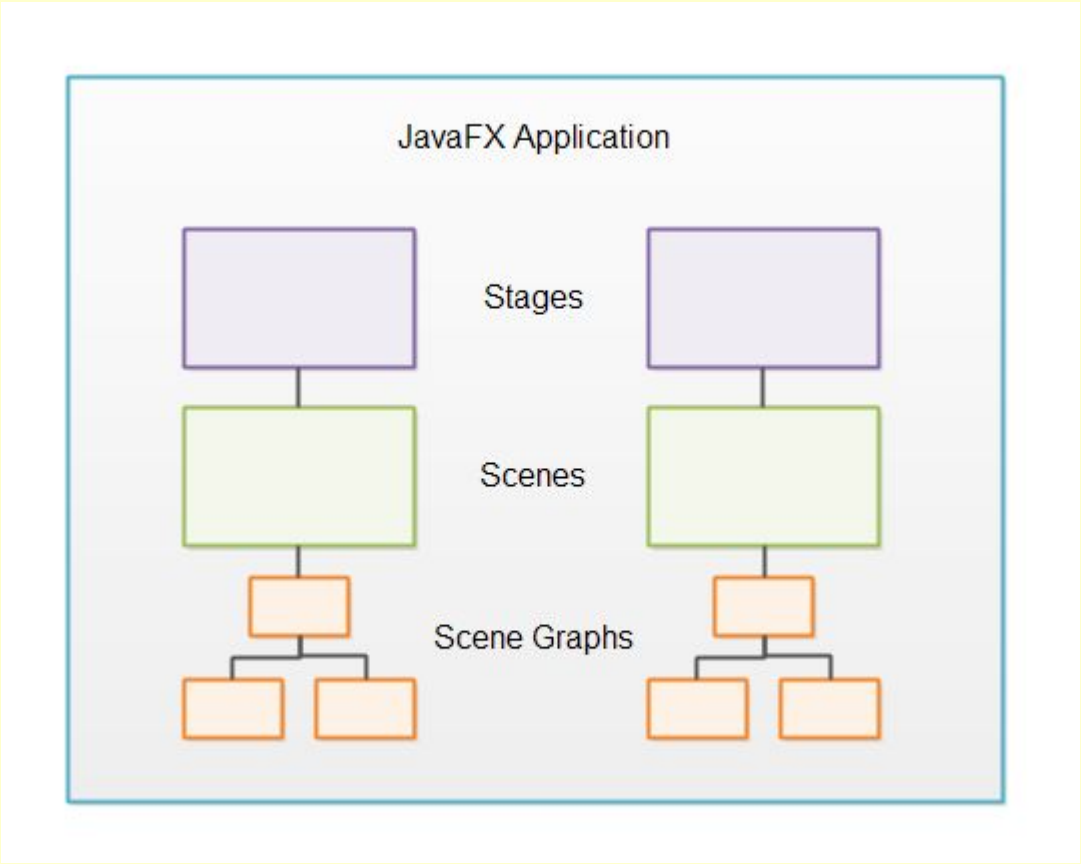
JavaFX can run on:

- Windows
- Linux
- Mac
- iOS
- Android

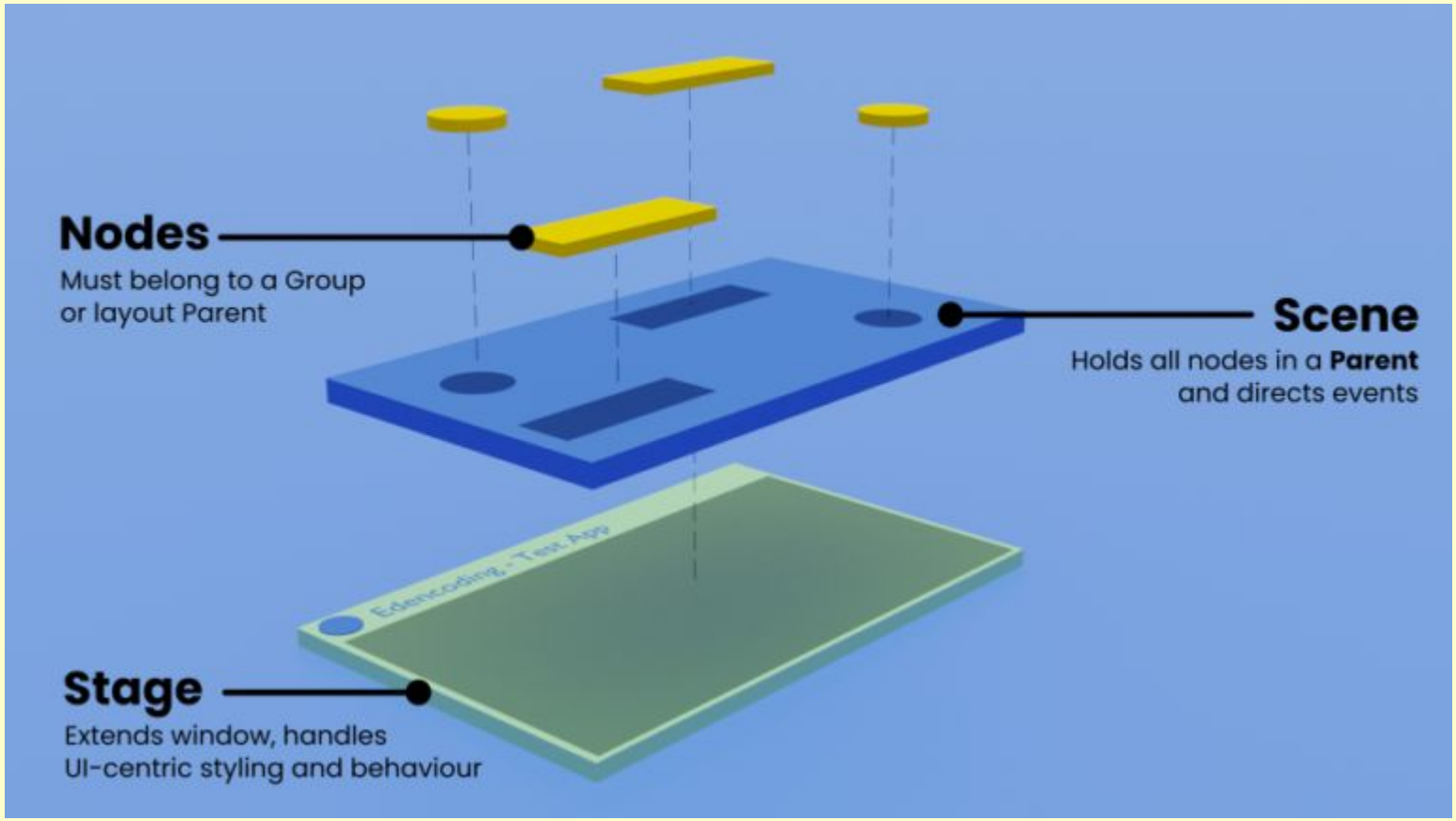
GUI components

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

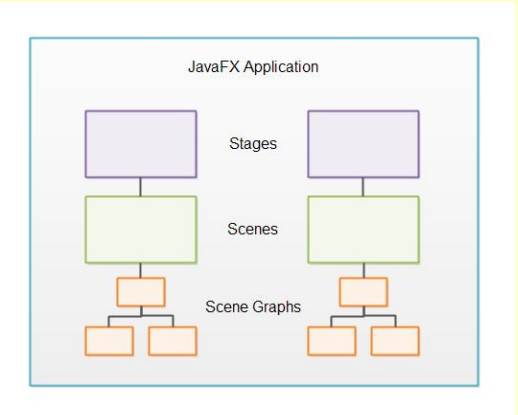


Stage vs Scene



source

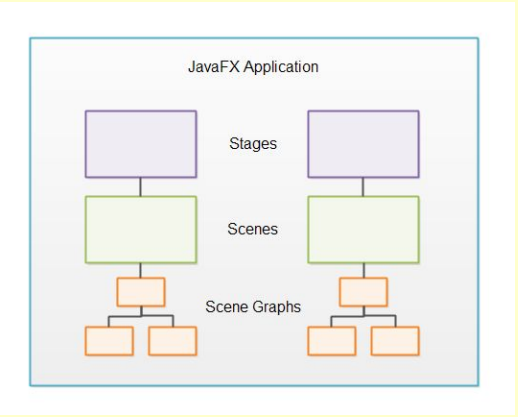
Stage



- outer frame (window)
- primary Stage object

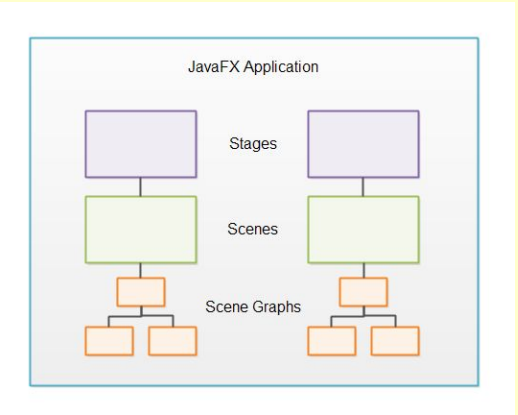
  created by the JavaFX runtime

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

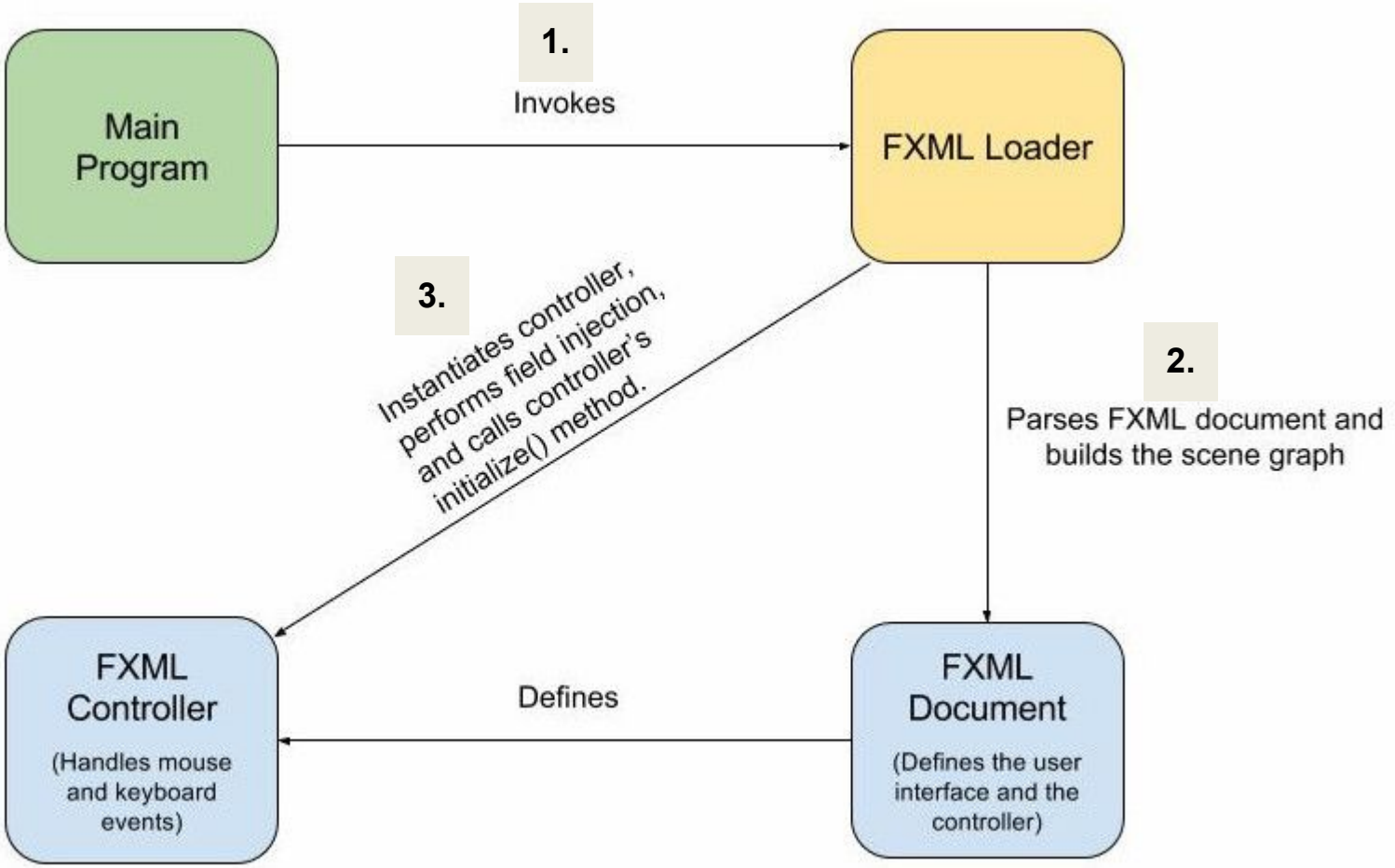
Your first JavaFX application

```
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

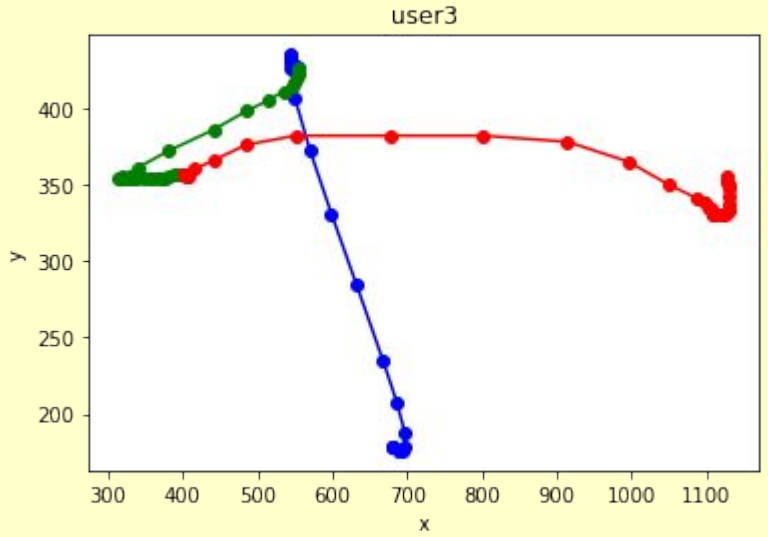
```
<TextField
    fx:id="inputText"
    prefWidth="100.0" />

<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

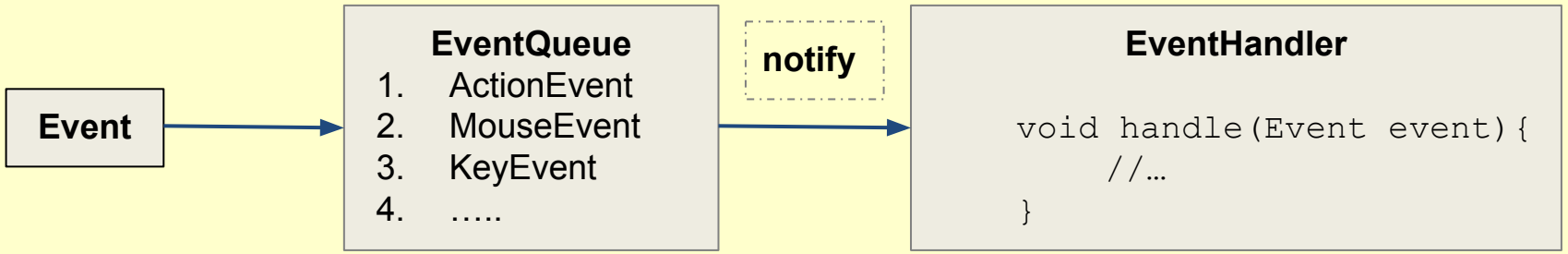
Types of events

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

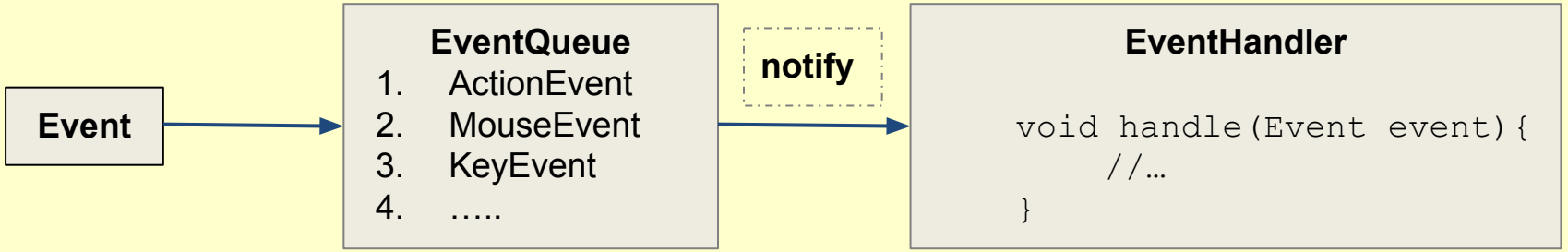
Event Driven Programming

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!*).

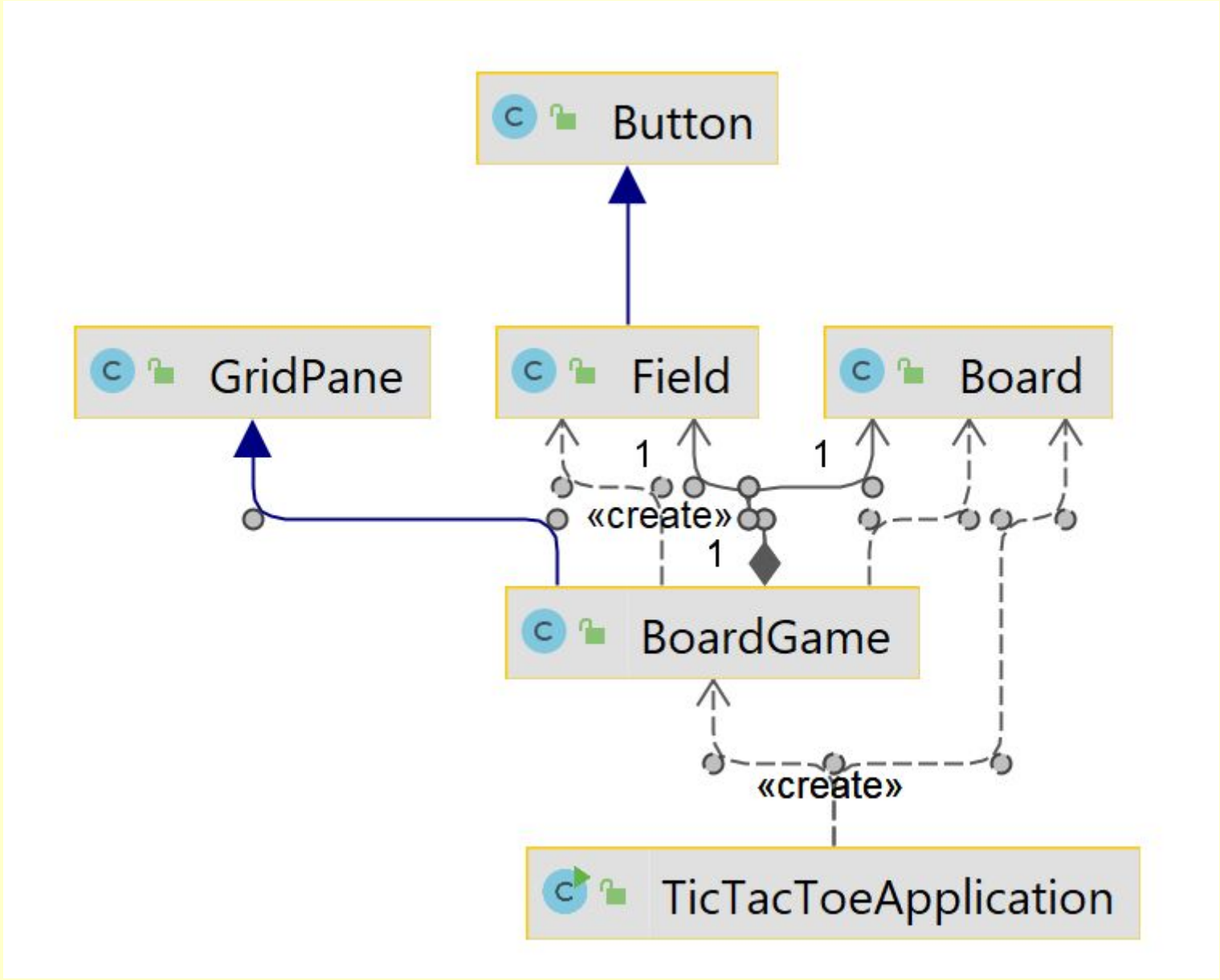
3. Write it as a **lambda expression** and use a reference variable to add it.

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 – resusable 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

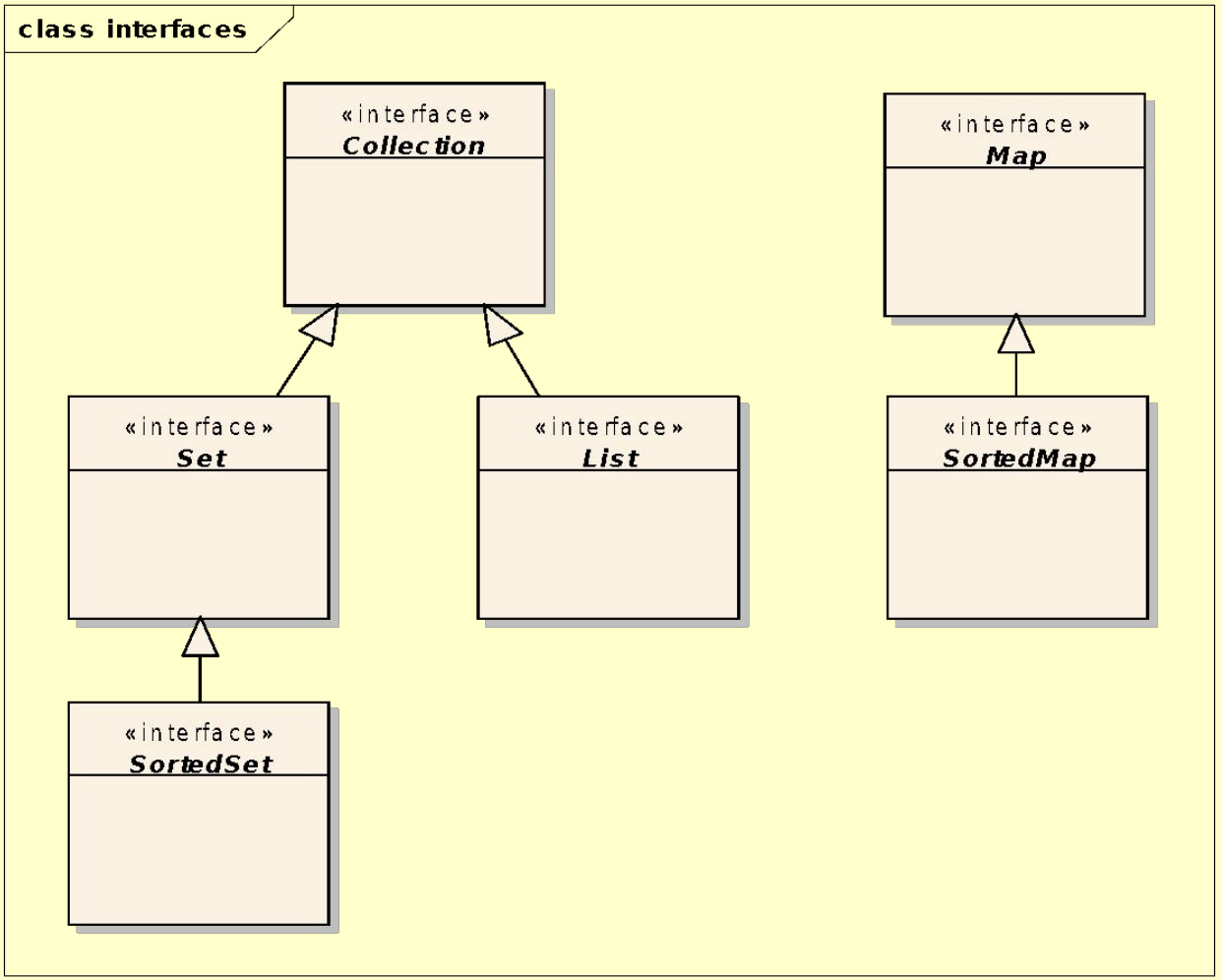
The Collections API

Design Goals

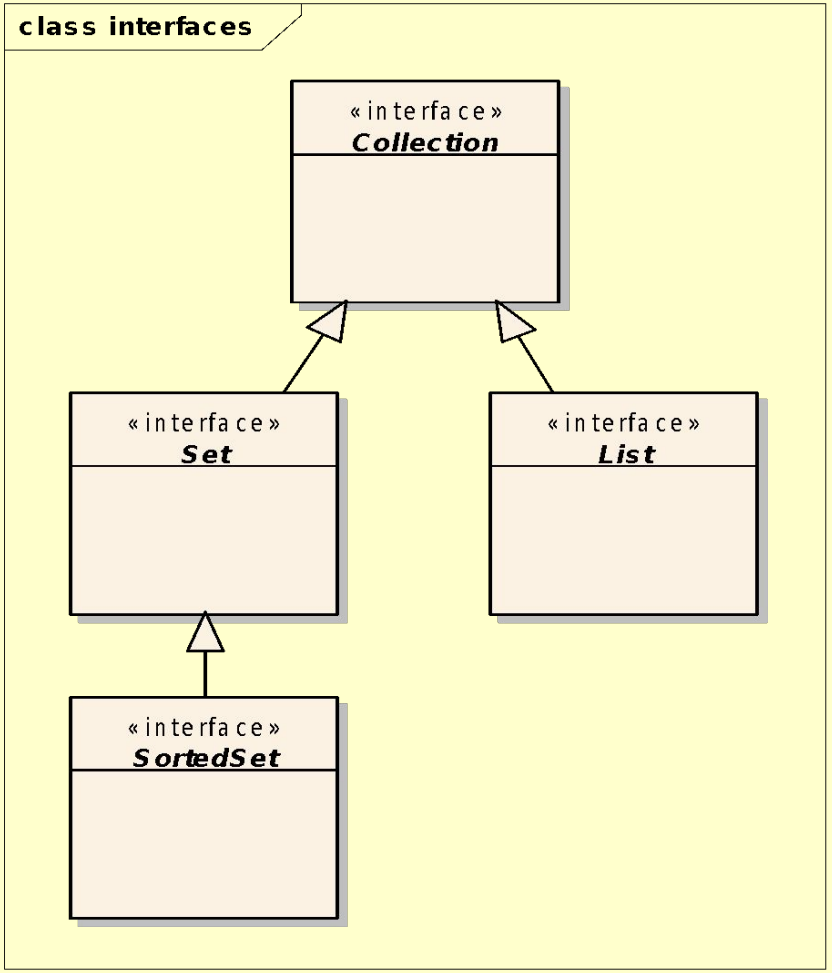
- Small and simple
- Powerful
- Easily extensible
- Compatible with preexisting collections
- Easy to use

The Collections API

Interfaces



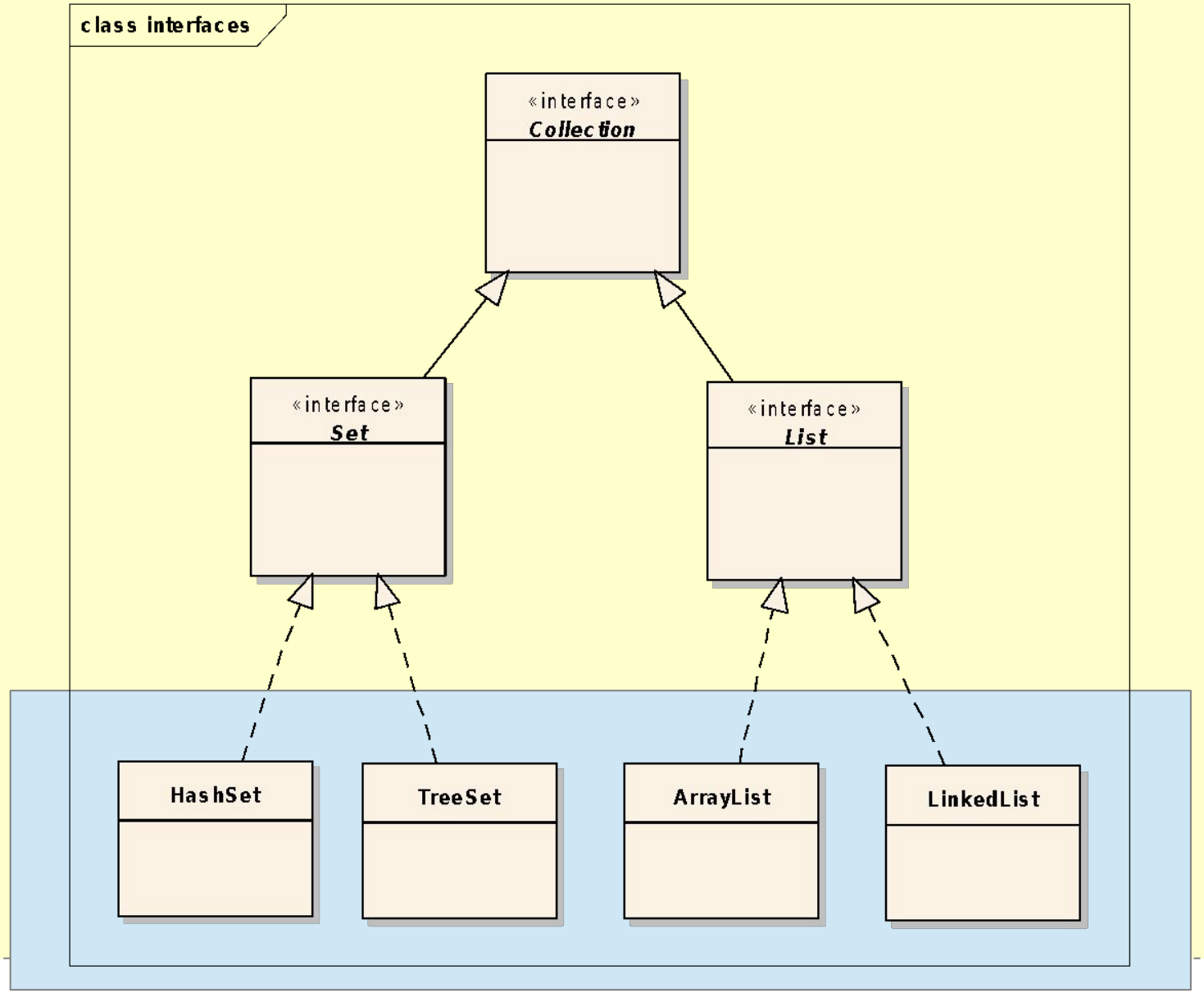
The `Collection` interface



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`

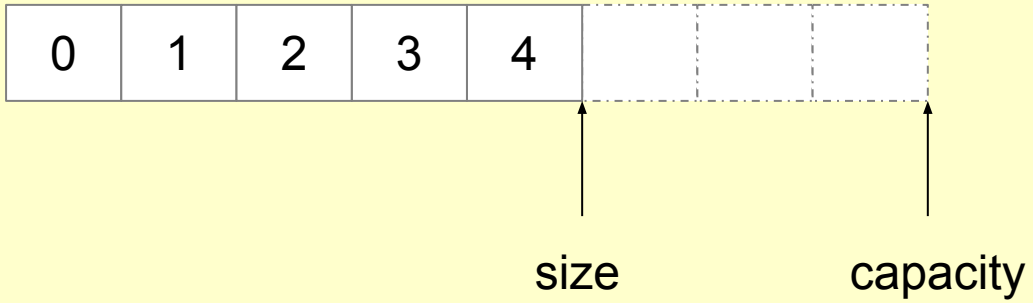
Implementations



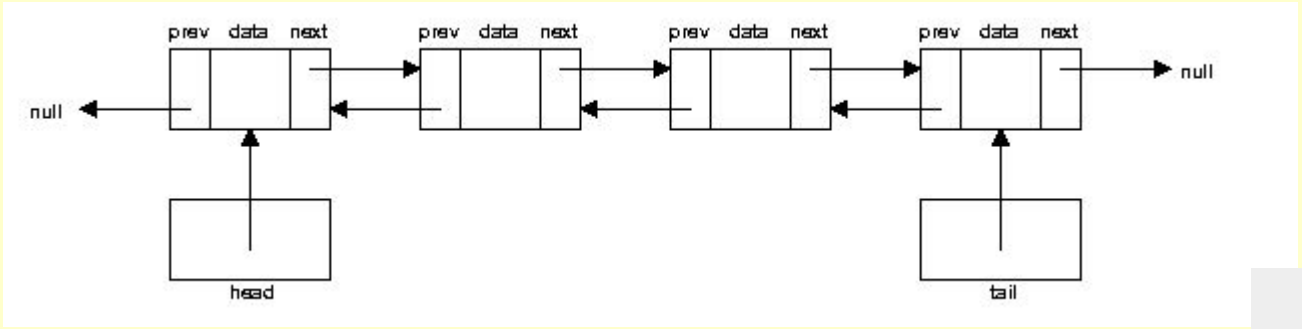
Implementations

`List` implementations

ArrayList



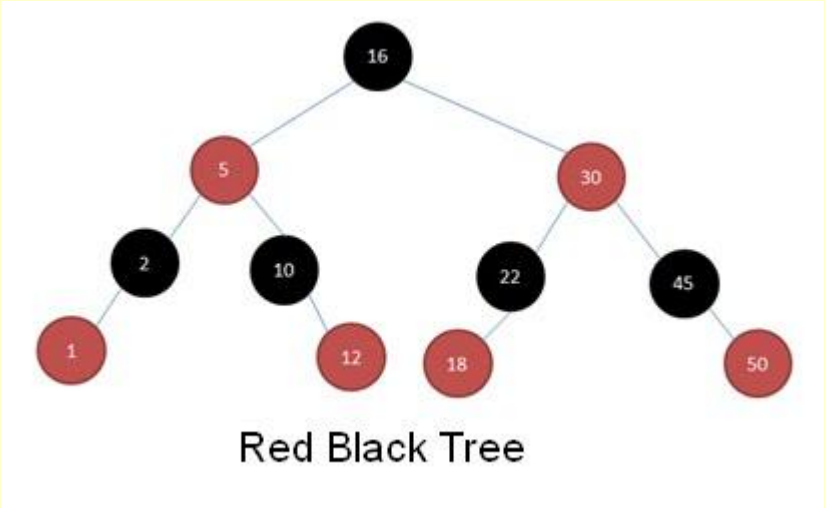
LinkedList



Source

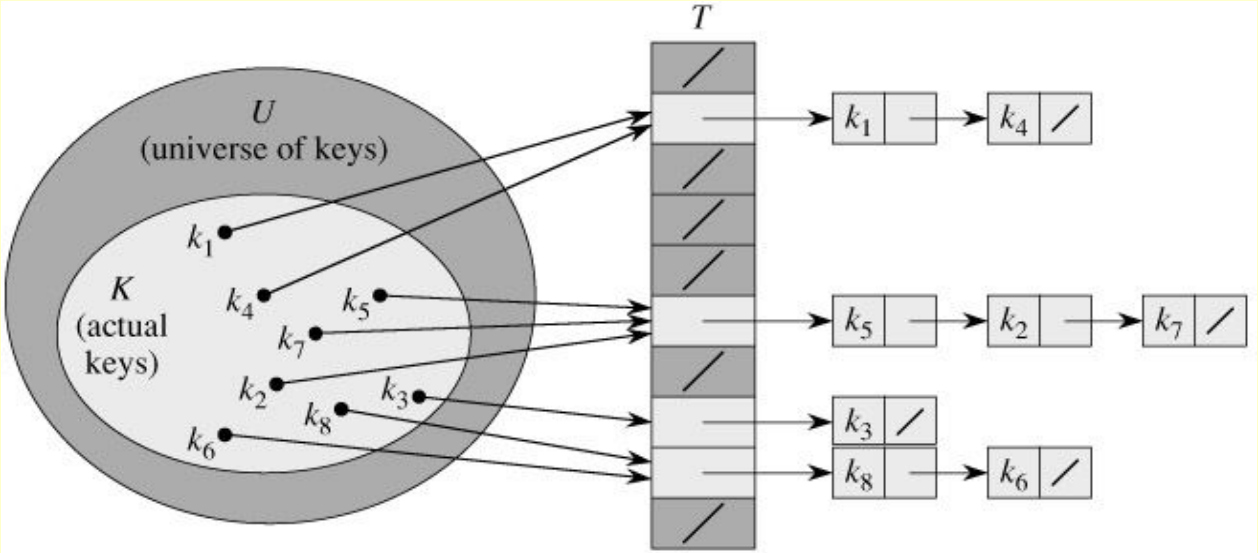
`Set` implementations

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 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 are ordered 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.

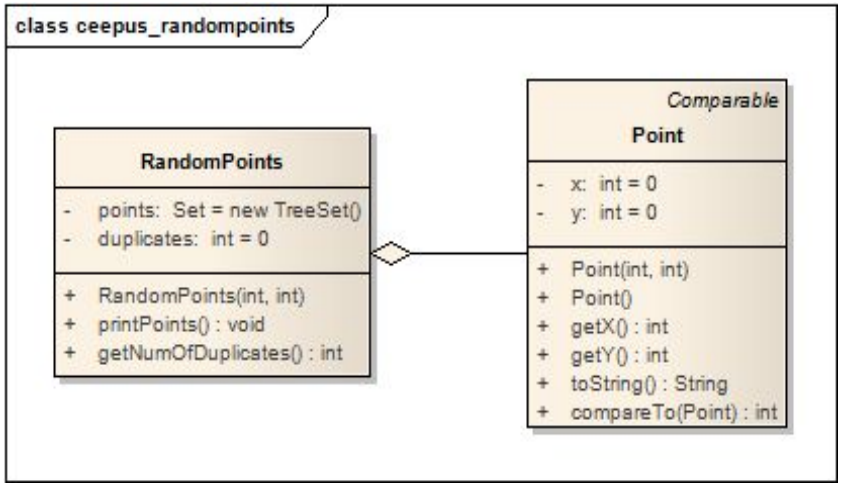
Which collection to use?

Hint: Finding an existing element must be fast.

Problem

1. solution - TreeSet

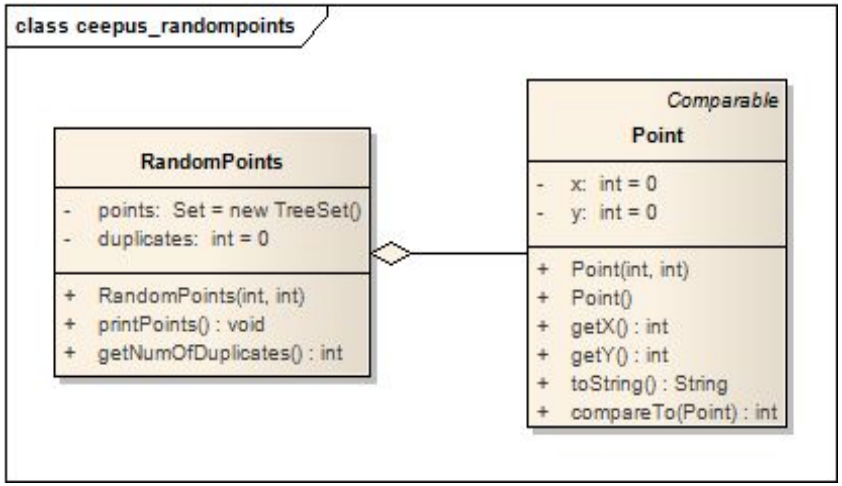
```
public class Point implements
                    Comparable<Point> {
   ...
    @Override
    public int compareTo(Point o) {
        if( o == null ) throw
           new NullPointerException();
        if (this.x == o.x &&
            this.y == o.y){
            return 0;
        }
        if(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



Problem

1. solution - TreeSet

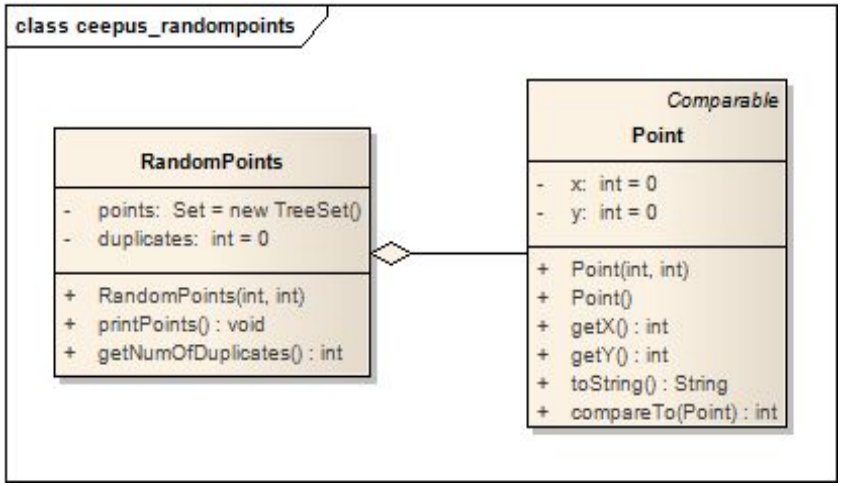
```
public class RandomPoints {
    private TreeSet<Point> points =
                    new TreeSet<Point>();
    private int duplicates = 0;

    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

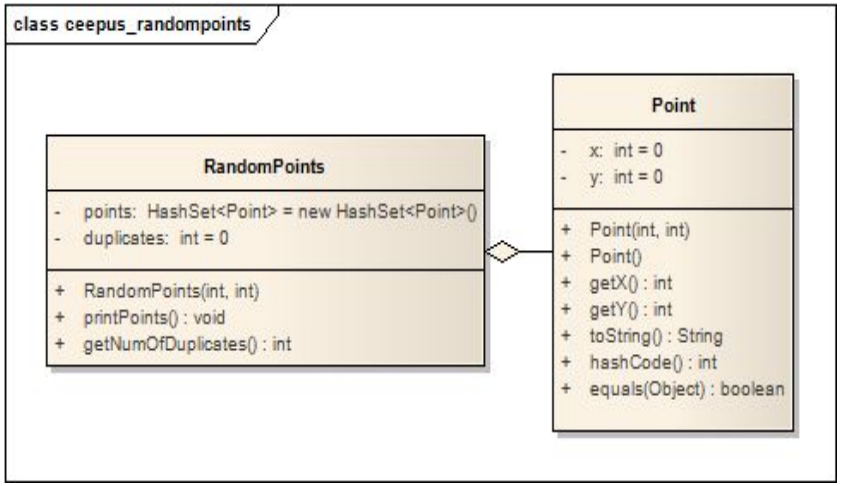
Duplicates: 4976

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

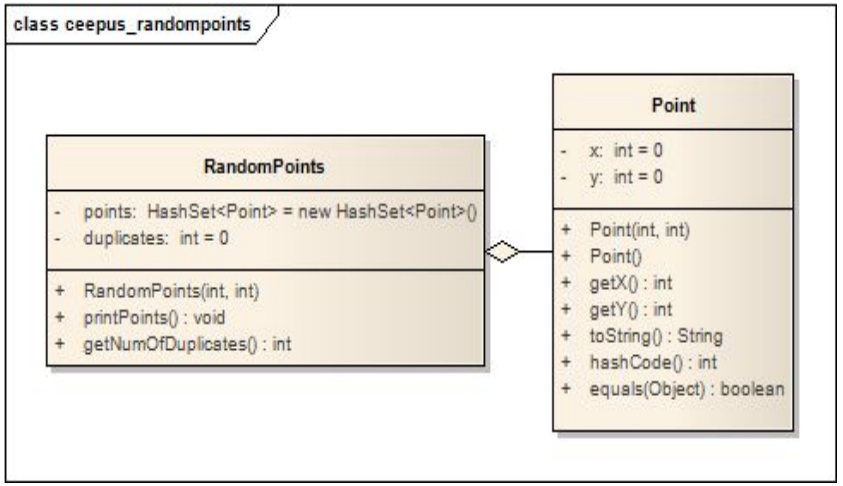
Random number generator: seed = 0
N = 1.000.000
M = 10.000
Duplicates: 4976
Time: **approx. 1 s**

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

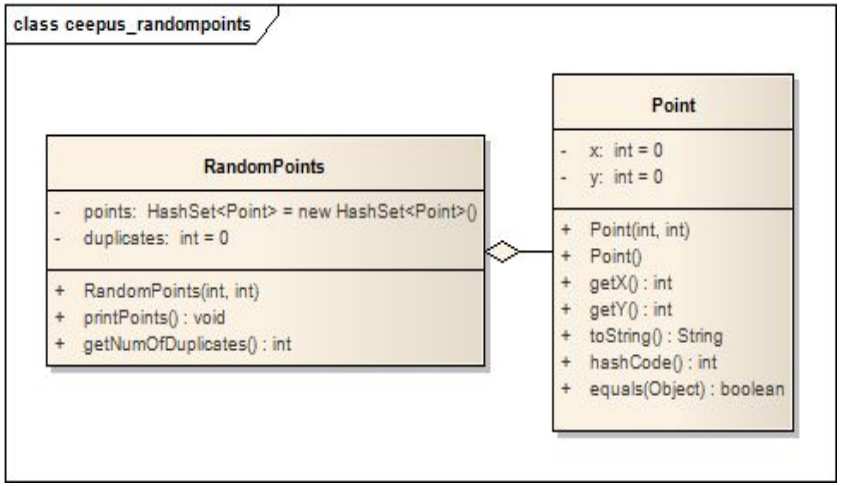
Random number generator: seed = 0
N = 1.000.000
M = 10.000
Duplicates: 4976
Time: **approx. 1s**

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?

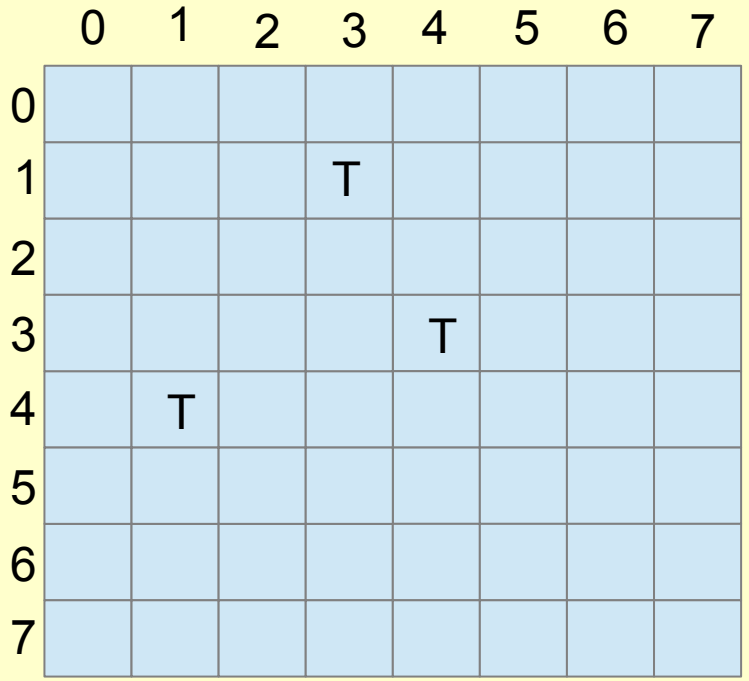
Problem

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;
   }
}
```



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

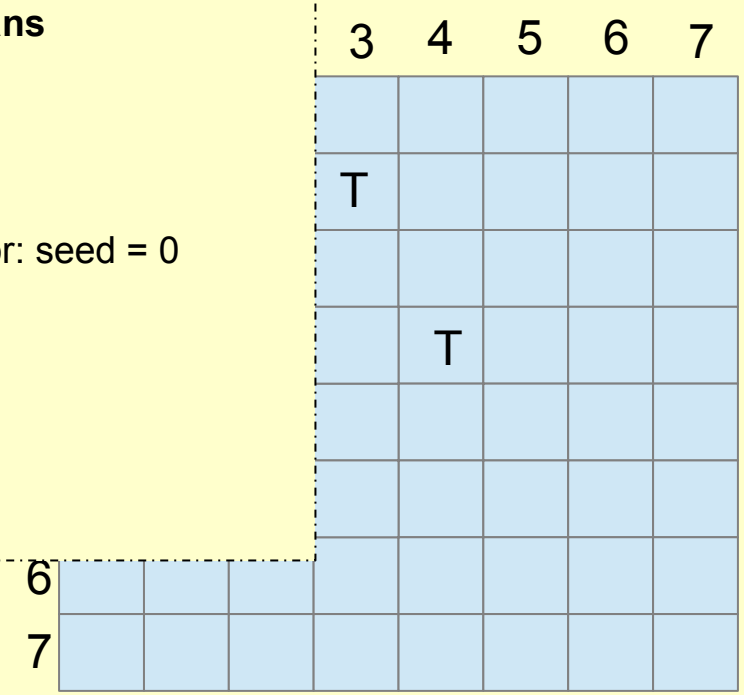
Random number generator: seed = 0

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.

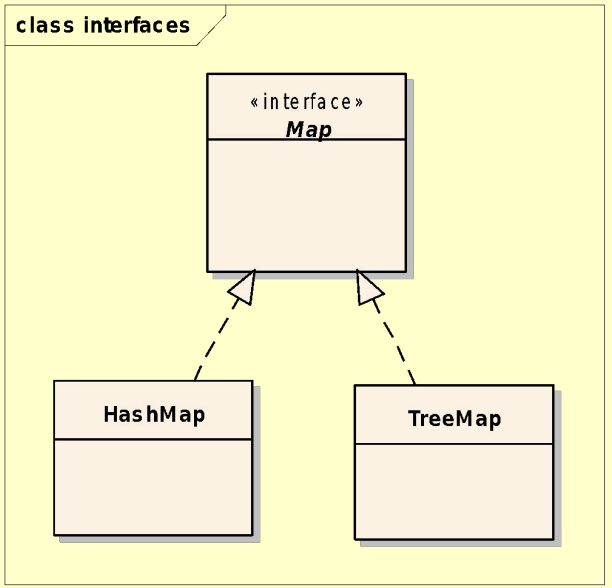
Examples:

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));
}
```

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

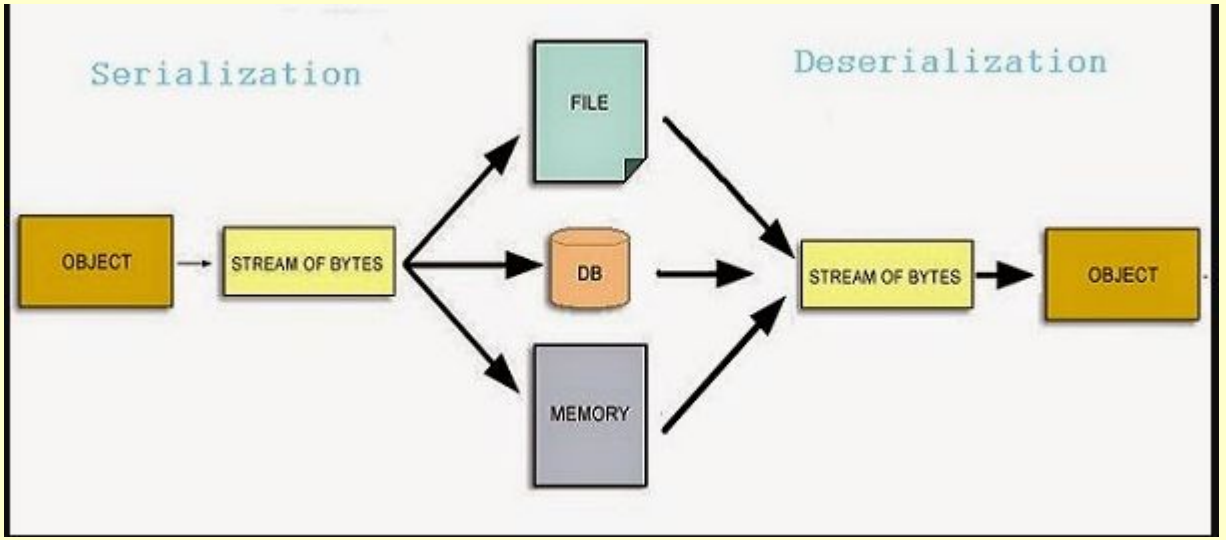
```
Map<String, Set<String> > groups = new HashMap<>();
//...

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

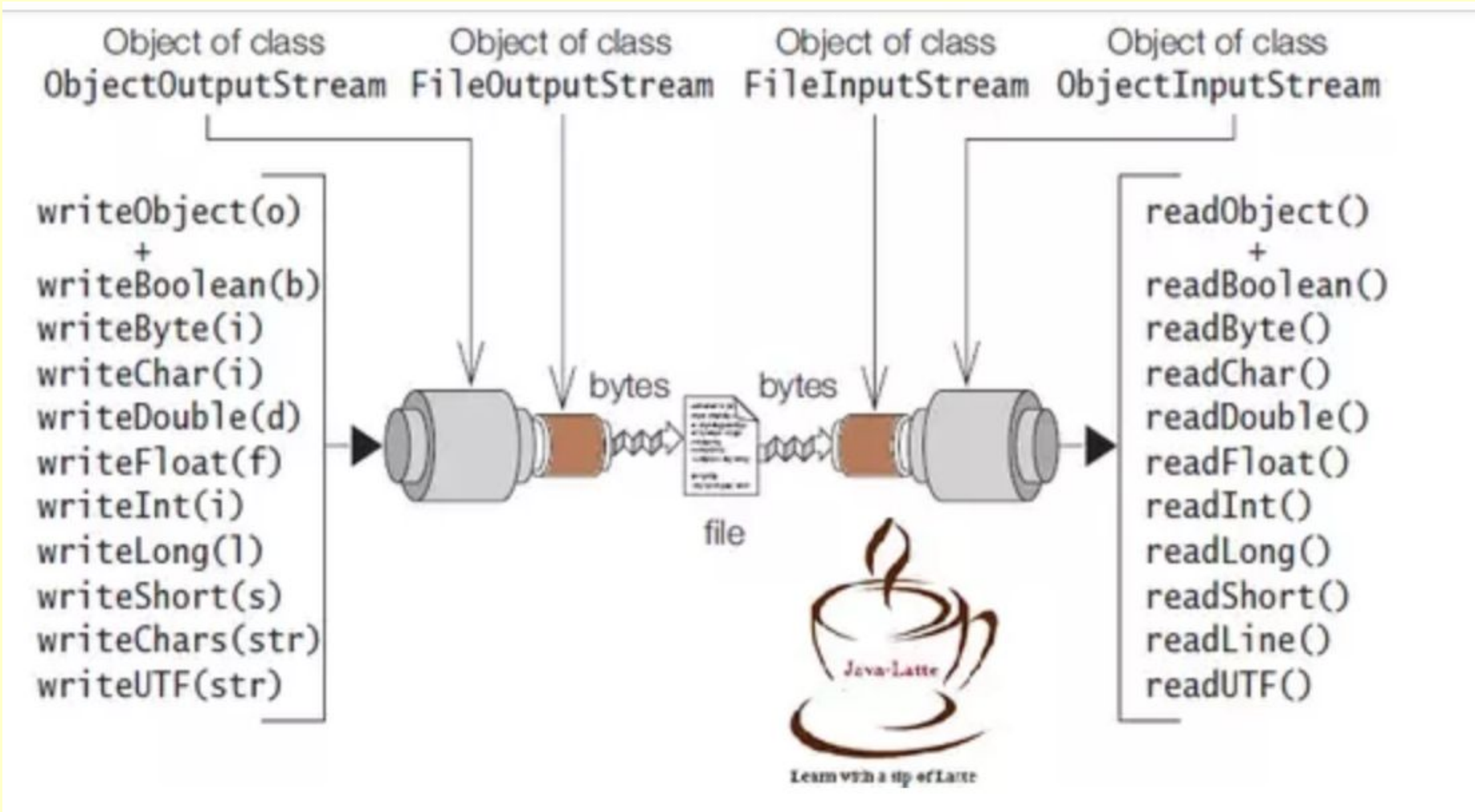
Anagrams

```
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();
}
```

Java AWT

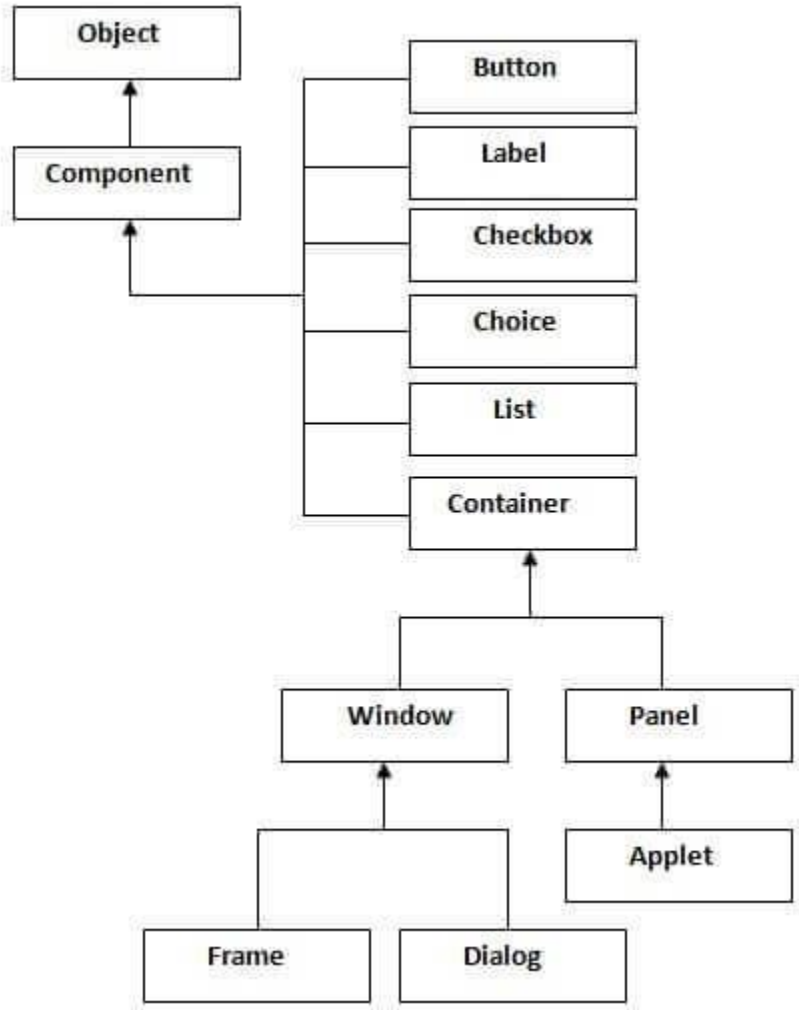
Java AWT (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Reference:- www.javatpoint.com and www.tutorialspoint.com

Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Reference:- www.javatpoint.com and www.tutorialspoint.com

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

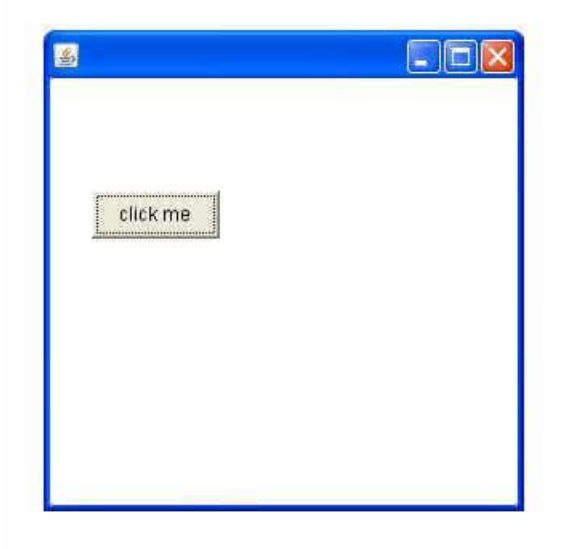
AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```
1. import java.awt.*;
2. class First extends Frame{
3. First(){
4. Button b=new Button("click me");
5. b.setBounds(30,100,80,30);// setting button position
6. add(b);//adding button into frame
7. setSize(300,300);//frame size 300 width and 300 height
8. setLayout(null);//no layout manager
9. setVisible(true);//now frame will be visible, by default not visible
10. }
11. public static void main(String args[]){
12. First f=new First();
13. }}
```

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.

Reference:- www.javatpoint.com and www.tutorialspoint.com

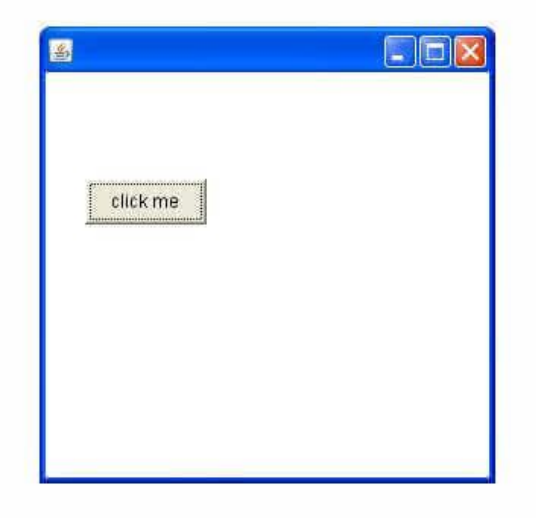


AWT Example by Association

Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are showing Button component on the Frame.

```
1. import java.awt.*;
2. class First2{
3. First2(){
4. Frame f=new Frame();
5. Button b=new Button("click me");
6. b.setBounds(30,50,80,30);
7. f.add(b);
8. f.setSize(300,300);
9. f.setLayout(null);
10. f.setVisible(true);
11. }
12. public static void main(String args[]){
13. First2 f=new First2();
14. }}
```

Reference:- www.javatpoint.com and www.tutorialspoint.com



Basic Terminologies

Term	Description
Component	Component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. For examples buttons, checkboxes, list and scrollbars of a graphical user interface.
Container	Container object is a component that can contain other components.Components added to a container are tracked in a list. The order of the list will define the components' front-to-back stacking order within the container. If no index is specified when adding a component to a container, it will be added to the end of the list.
Panel	Panel provides space in which an application can attach any other components, including other panels.
Window	Window is a rectangular area which is displayed on the screen. In different window we can execute different program and display different data. Window provide us with multitasking environment. A window must have either a frame, dialog, or another window defined as its owner when it's constructed.
Frame	A Frame is a top-level window with a title and a border. The size of the frame includes any area designated for the border. Frame encapsulates **window**. It and has a title bar, menu bar, borders, and resizing corners.

Reference:- www.javatpoint.com and www.tutorialspoint.com

Canvas	Canvas component represents a blank rectangular area of the screen onto which the application can draw. Application can also trap input events from the use from that blank area of Canvas component.

Examples of GUI based Applications

Following are some of the examples for GUI based applications.

- Automated Teller Machine (ATM)
- Airline Ticketing System
- Information Kiosks at railway stations
- Mobile Applications
- Navigation Systems

AWT Controls

Every user interface considers the following three main aspects:

- **UI elements** : Thes are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex which we will cover in this tutorial.
- **Layouts:** They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in Layout chapter.
- **Behavior:** These are events which occur when the user interacts with UI elements. This part will be covered in Event Handling chapter.

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc
package provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener

Reference:- www.javatpoint.com and www.tutorialspoint.com

MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener

Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**

Reference:- www.javatpoint.com and www.tutorialspoint.com

- public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

Java Event Handling Code

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

Java event handling by implementing ActionListener

```
1. import java.awt.*;
2. import java.awt.event.*;
3. class AEvent extends Frame implements ActionListener{
4. TextField tf;
5. AEvent(){
6. 
7. //create components
8. tf=new TextField();
9. tf.setBounds(60,50,170,20);
10. Button b=new Button("click me");
11. b.setBounds(100,120,80,30);
12. 
13. //register listener
14. b.addActionListener(this);//passing current instance
15. 
16. //add components and set size, layout and visibility
17. add(b);add(tf);
18. setSize(300,300);
19. setLayout(null);
20. setVisible(true);
21. }
22. public void actionPerformed(ActionEvent e){
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
23. tf.setText("Welcome");
24. }
25. public static void main(String args[]){
26. new AEvent();
27. }
28. }
```

public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.

---

2) Java event handling by outer class

```
1. import java.awt.*;
2. import java.awt.event.*;
3. class AEvent2 extends Frame{
4. TextField tf;
5. AEvent2(){
6. //create components
7. tf=new TextField();
8. tf.setBounds(60,50,170,20);
9. Button b=new Button("click me");
10. b.setBounds(100,120,80,30);
11. //register listener
12. Outer o=new Outer(this);
13. b.addActionListener(o);//passing outer class instance
14. //add components and set size, layout and visibility
15. add(b);add(tf);
16. setSize(300,300);
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
17. setLayout(null);
18. setVisible(true);
19. }
20. public static void main(String args[]){
21. new AEvent2();
22. }
23. }
```

```
1.  import java.awt.event.*;
2.  class Outer implements ActionListener{
3.  AEvent2 obj;
4.  Outer(AEvent2 obj){
5.  this.obj=obj;
6.  }
7.  public void actionPerformed(ActionEvent e){
8.  obj.tf.setText("welcome");
9.  }
10. }
```

---

3) Java event handling by anonymous class

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  class AEvent3 extends Frame{
4.  TextField tf;
5.  AEvent3(){
6.  tf=new TextField();
7.  tf.setBounds(60,50,170,20);
8.  Button b=new Button("click me");
9.  b.setBounds(50,120,80,30);
10.
11. b.addActionListener(new ActionListener(){
12. public void actionPerformed(){
13. tf.setText("hello");
14. }
15. });
16. add(b);add(tf);
17. setSize(300,300);
18. setLayout(null);
19. setVisible(true);
20. }
21. public static void main(String args[]){
22. new AEvent3();
23. }
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
24. }
```

Every AWT controls inherits properties from Component class.

Sr. No.	Control & Description
1	Component A Component is an abstract super class for GUI controls and it represents an object with graphical representation.

AWT UI Elements:

Following is the list of commonly used controls while designed GUI using AWT.

Sr. No.	Control & Description
1	Label A Label object is a component for placing text in a container.
2	Button This class creates a labeled button.
3	Check Box A check box is a graphical component that can be in either an **on** (true) or **off** (false) state.
4	Check Box Group The CheckboxGroup class is used to group the set of checkbox.
5	List The List component presents the user with a scrolling list of text items.
6	Text Field A TextField object is a text component that allows for the editing of a single line of text.

Reference:- www.javatpoint.com and www.tutorialspoint.com

7	Text Area A TextArea object is a text component that allows for the editing of a multiple lines of text.
8	Choice A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.
9	Canvas A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.
10	Image An Image control is superclass for all image classes representing graphical images.
11	Scroll Bar A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
12	Dialog A Dialog control represents a top-level window with a title and a border used to take some form of input from the user.
13	File Dialog A FileDialog control represents a dialog window from which the user can select a file.

AWT Event Handling

What is an Event?

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

Points to remember about listener

- In order to design a listener class we have to develop some listener interfaces.These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.
- If you do not implement the any if the predefined interfaces then your class can not act as a listener class for a source object.

Callback Methods

Reference:- www.javatpoint.com and www.tutorialspoint.com

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire callback method. All such callback methods are provided in listener interfaces.

If a component wants some listener will listen to it's events the the source must register itself to the listener.

AWT Event Classes

The Event classes represent the event. Java provides us various Event classes but we will discuss those which are more frequently used.

EventObject class

It is the root class from which all event state objects shall be derived. All Events are constructed with a reference to the object, the **source**, that is logically deemed to be the object upon which the Event in question initially occurred upon.This class is defined in java.util package.

Class declaration

Following is the declaration for **java.util.EventObject** class:

```
public class EventObject
   extends Object
      implements Serializable
```

Field

Following are the fields for **java.util.EventObject** class:

- **protected Object source** -- The object on which the Event initially occurred.

Class constructors

S.N.	Constructor & Description
1	**EventObject(Object source)** Constructs a prototypical Event.

Class methods

S.N.	Method & Description

Reference:- www.javatpoint.com and www.tutorialspoint.com

1	**Object getSource()** The object on which the Event initially occurred.
2	**String toString()** Returns a String representation of this EventObject.

Methods inherited

This class inherits methods from the following classes:

- java.lang.Object

AWT Event Classes:

Following is the list of commonly used event classes.

Sr. No.	Control & Description
1	AWTEvent It is the root event class for all AWT events. This class and its subclasses supercede the original java.awt.Event class.
2	ActionEvent The ActionEvent is generated when button is clicked or the item of a list is double clicked.
3	InputEvent The InputEvent class is root event class for all component-level input events.
4	KeyEvent On entering the character the Key event is generated.
5	MouseEvent This event indicates a mouse action occurred in a component.
6	TextEvent The object of this class represents the text events.

Reference:- www.javatpoint.com and www.tutorialspoint.com

7	WindowEvent The object of this class represents the change in state of a window.
8	AdjustmentEvent The object of this class represents the adjustment event emitted by Adjustable objects.
9	ComponentEvent The object of this class represents the change in state of a window.
10	ContainerEvent The object of this class represents the change in state of a window.
11	MouseMotionEvent The object of this class represents the change in state of a window.
12	PaintEvent The object of this class represents the change in state of a window.

Java ActionListener Interface

The Java ActionListener is notified whenever you click on the button or menu item. It is notified against ActionEvent. The ActionListener interface is found in java.awt.event package. It has only one method: actionPerformed().

actionPerformed() method

The actionPerformed() method is invoked automatically whenever you click on the registered component.

```
1. public abstract void actionPerformed(ActionEvent e);
```

How to write ActionListener

The common approach is to implement the ActionListener. If you implement the ActionListener class, you need to follow 3 steps:

1) Implement the ActionListener interface in the class:

```
1. public class ActionListenerExample Implements ActionListener
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

2) Register the component with the Listener:

```
component.addActionListener(instanceOfListenerclass);
```

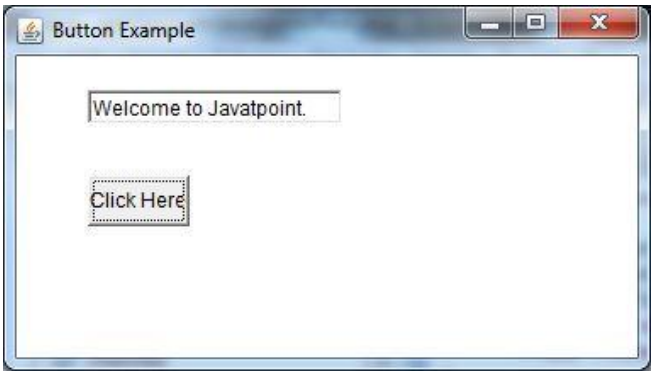
3) Override the actionPerformed() method:

```
public void actionPerformed(ActionEvent e){
        //Write the code here
}
```

Java ActionListener Example: On Button click

```
import java.awt.*;
import java.awt.event.*;
//1st step
public class ActionListenerExample implements ActionListener{
public static void main(String[] args) {
    Frame f=new Frame("ActionListener Example");
    final TextField tf=new TextField();
    tf.setBounds(50,50, 150,20);
    Button b=new Button("Click Here");
    b.setBounds(50,100,60,30);
    //2nd step
    b.addActionListener(this);
    f.add(b);f.add(tf);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
//3rd step
public void actionPerformed(ActionEvent e){
        tf.setText("Welcome to Javatpoint.");
}
}
```

Output:



Java ActionListener Example: Using Anonymous class

We can also use the anonymous class to implement the ActionListener. It is the shorthand way, so you do not need to follow the 3 steps:

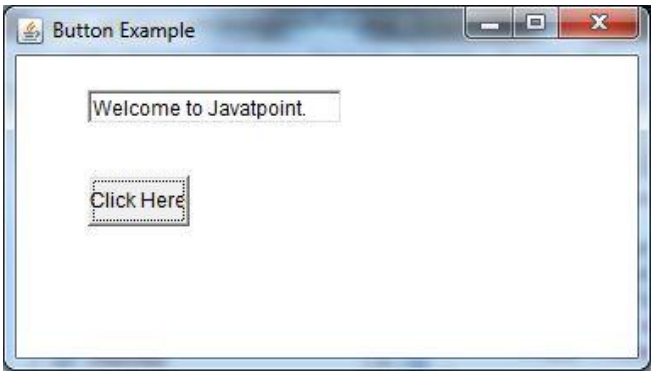
```
1. b.addActionListener(new ActionListener(){
2. public void actionPerformed(ActionEvent e){
3.          tf.setText("Welcome to Javatpoint.");
4. }
5. });
```

Let us see the full code of ActionListener using anonymous class.

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class ActionListenerExample {
4. public static void main(String[] args) {
5.     Frame f=new Frame("ActionListener Example");
6.     final TextField tf=new TextField();
7.     tf.setBounds(50,50, 150,20);
8.     Button b=new Button("Click Here");
9.     b.setBounds(50,100,60,30);
10.
11.    b.addActionListener(new ActionListener(){
12.    public void actionPerformed(ActionEvent e){
13.          tf.setText("Welcome to Javatpoint.");
14.    }
15.    });
16.    f.add(b);f.add(tf);
17.    f.setSize(400,400);
18.    f.setLayout(null);
19.    f.setVisible(true);
20. }
21. }
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

Output:



Java AWT Label

The object of Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly.

AWT Label Class Declaration

```
1. public class Label extends Component implements Accessible
```

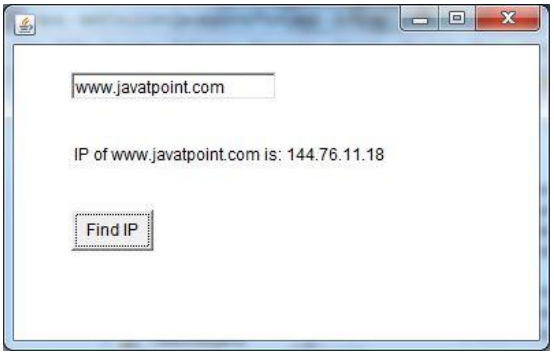
Java AWT Label Example with ActionListener

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class LabelExample extends Frame implements ActionListener{
4.     TextField tf; Label l; Button b;
5.     LabelExample(){
6.         tf=new TextField();
7.         tf.setBounds(50,50, 150,20);
8.         l=new Label();
9.         l.setBounds(50,100, 250,20);
10.        b=new Button("Find IP");
11.        b.setBounds(50,150,60,30);
12.        b.addActionListener(this);
13.        add(b);add(tf);add(l);
14.        setSize(400,400);
15.        setLayout(null);
16.        setVisible(true);
17.    }
18.    public void actionPerformed(ActionEvent e) {
19.        try{
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
20.         String host=tf.getText();
21.         String ip=java.net.InetAddress.getByName(host).getHostAddress();
22.         l.setText("IP of "+host+" is: "+ip);
23.         }catch(Exception ex){System.out.println(ex);}
24.     }
25.     public static void main(String[] args) {
26.         new LabelExample();
27.     }
28. }
```

Output:



Java AWT TextField

The object of a TextField class is a text component that allows the editing of a single line text. It inherits TextComponent class.

AWT TextField Class Declaration

```
1. public class TextField extends TextComponent
```

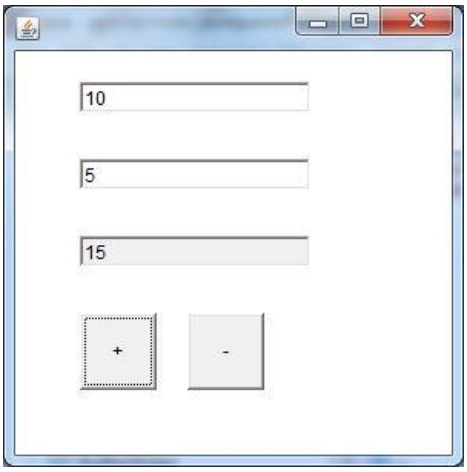
Java AWT TextField Example with ActionListener

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class TextFieldExample extends Frame implements ActionListener{
4.     TextField tf1,tf2,tf3;
5.     Button b1,b2;
6.     TextFieldExample(){
7.         tf1=new TextField();
8.         tf1.setBounds(50,50,150,20);
9.         tf2=new TextField();
10.        tf2.setBounds(50,100,150,20);
11.        tf3=new TextField();
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
12.         tf3.setBounds(50,150,150,20);
13.         tf3.setEditable(false);
14.         b1=new Button("+");
15.         b1.setBounds(50,200,50,50);
16.         b2=new Button("-");
17.         b2.setBounds(120,200,50,50);
18.         b1.addActionListener(this);
19.         b2.addActionListener(this);
20.         add(tf1);add(tf2);add(tf3);add(b1);add(b2);
21.         setSize(300,300);
22.         setLayout(null);
23.         setVisible(true);
24.     }
25.     public void actionPerformed(ActionEvent e) {
26.         String s1=tf1.getText();
27.         String s2=tf2.getText();
28.         int a=Integer.parseInt(s1);
29.         int b=Integer.parseInt(s2);
30.         int c=0;
31.         if(e.getSource()==b1){
32.             c=a+b;
33.         }else if(e.getSource()==b2){
34.             c=a-b;
35.         }
36.         String result=String.valueOf(c);
37.         tf3.setText(result);
38.     }
39. public static void main(String[] args) {
40.     new TextFieldExample();
41. }
42. }
```

Output:



Java AWT TextArea

The object of a TextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

AWT TextArea Class Declaration

```
1. public class TextArea extends TextComponent
```

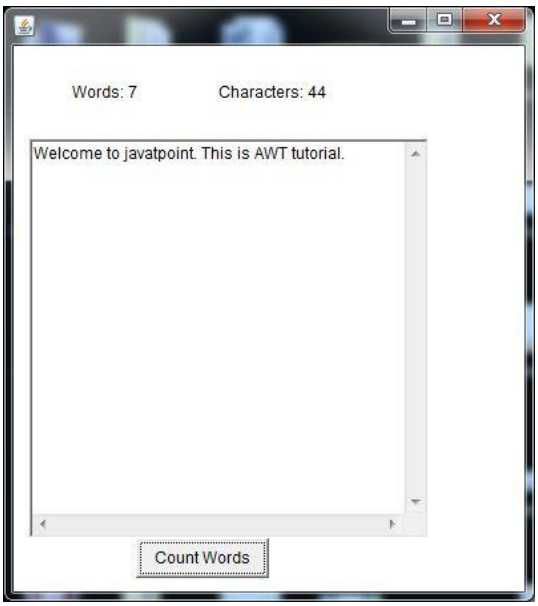
Java AWT TextArea Example with ActionListener

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class TextAreaExample extends Frame implements ActionListener{
4. Label l1,l2;
5. TextArea area;
6. Button b;
7. TextAreaExample(){
8.     l1=new Label();
9.     l1.setBounds(50,50,100,30);
10.    l2=new Label();
11.    l2.setBounds(160,50,100,30);
12.    area=new TextArea();
13.    area.setBounds(20,100,300,300);
14.    b=new Button("Count Words");
15.    b.setBounds(100,400,100,30);
16.    b.addActionListener(this);
17.    add(l1);add(l2);add(area);add(b);
18.    setSize(400,450);
19.    setLayout(null);
20.    setVisible(true);
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
21. }
22. public void actionPerformed(ActionEvent e){
23.     String text=area.getText();
24.     String words[]=text.split("\\s");
25.     l1.setText("Words: "+words.length);
26.     l2.setText("Characters: "+text.length());
27. }
28. public static void main(String[] args) {
29.     new TextAreaExample();
30. }
31. }
```

Output:



Java MouseListener Interface

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface

The signature of 5 methods found in MouseListener interface are given below:

1. `public abstract void mouseClicked(MouseEvent e);`
2. `public abstract void mouseEntered(MouseEvent e);`

Reference:- www.javatpoint.com and www.tutorialspoint.com

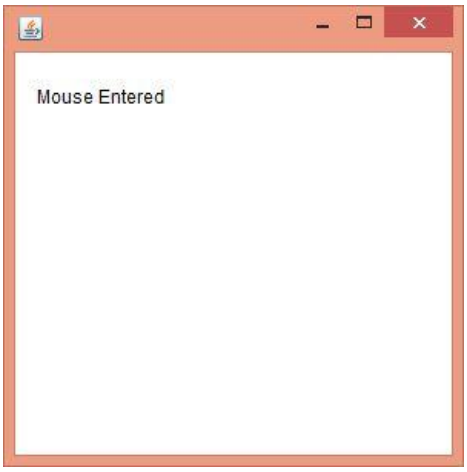
```
3. public abstract void mouseExited(MouseEvent e);
4. public abstract void mousePressed(MouseEvent e);
5. public abstract void mouseReleased(MouseEvent e);
```

Java MouseListener Example

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class MouseListenerExample extends Frame implements MouseListener{
4.     Label l;
5.     MouseListenerExample(){
6.         addMouseListener(this);
7.
8.         l=new Label();
9.         l.setBounds(20,50,100,20);
10.        add(l);
11.        setSize(300,300);
12.        setLayout(null);
13.        setVisible(true);
14.    }
15.    public void mouseClicked(MouseEvent e) {
16.        l.setText("Mouse Clicked");
17.    }
18.    public void mouseEntered(MouseEvent e) {
19.        l.setText("Mouse Entered");
20.    }
21.    public void mouseExited(MouseEvent e) {
22.        l.setText("Mouse Exited");
23.    }
24.    public void mousePressed(MouseEvent e) {
25.        l.setText("Mouse Pressed");
26.    }
27.    public void mouseReleased(MouseEvent e) {
28.        l.setText("Mouse Released");
29.    }
30. public static void main(String[] args) {
31.    new MouseListenerExample();
32. }
33. }
```

Output:

Reference:- www.javatpoint.com and www.tutorialspoint.com

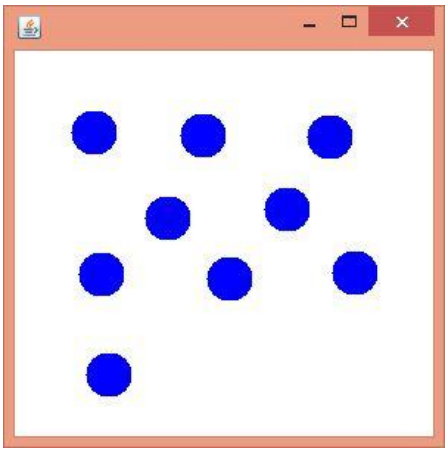


Java MouseListener Example 2

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class MouseListenerExample2 extends Frame implements MouseListener{
4.     MouseListenerExample2(){
5.         addMouseListener(this);
6.
7.         setSize(300,300);
8.         setLayout(null);
9.         setVisible(true);
10.    }
11.    public void mouseClicked(MouseEvent e) {
12.        Graphics g=getGraphics();
13.        g.setColor(Color.BLUE);
14.        g.fillOval(e.getX(),e.getY(),30,30);
15.    }
16.    public void mouseEntered(MouseEvent e) {}
17.    public void mouseExited(MouseEvent e) {}
18.    public void mousePressed(MouseEvent e) {}
19.    public void mouseReleased(MouseEvent e) {}
20.
21. public static void main(String[] args) {
22.    new MouseListenerExample2();
23. }
24. }
```

Output:

Reference:- www.javatpoint.com and www.tutorialspoint.com



Java MouseMotionListener Interface

The Java MouseMotionListener is notified whenever you move or drag mouse. It is notified against MouseEvent. The MouseMotionListener interface is found in java.awt.event package. It has two methods.

Methods of MouseMotionListener interface

The signature of 2 methods found in MouseMotionListener interface are given below:

```
1. public abstract void mouseDragged(MouseEvent e);
2. public abstract void mouseMoved(MouseEvent e);
```

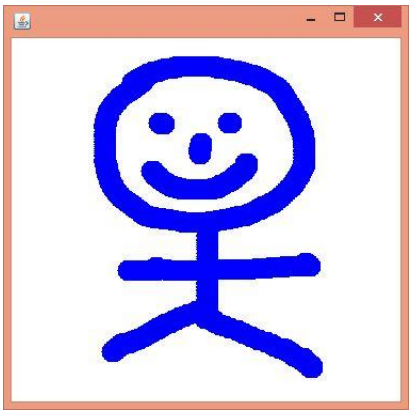
Java MouseMotionListener Example

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class MouseMotionListenerExample extends Frame implements MouseMotionListener{
4.     MouseMotionListenerExample(){
5.         addMouseMotionListener(this);
6.
7.         setSize(300,300);
8.         setLayout(null);
9.         setVisible(true);
10.    }
11. public void mouseDragged(MouseEvent e) {
12.    Graphics g=getGraphics();
13.    g.setColor(Color.BLUE);
14.    g.fillOval(e.getX(),e.getY(),20,20);
15. }
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
16. public void mouseMoved(MouseEvent e) {}
17.
18. public static void main(String[] args) {
19.     new MouseMotionListenerExample();
20. }
21. }
```

Output:



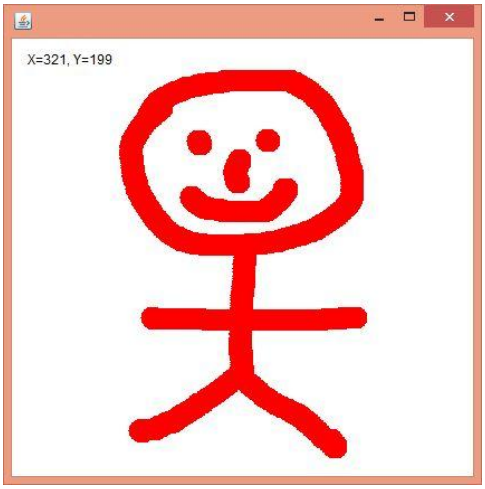
Java MouseMotionListener Example 2

```
1.  import java.awt.*;
2.  import java.awt.event.MouseEvent;
3.  import java.awt.event.MouseMotionListener;
4.  public class Paint extends Frame implements MouseMotionListener{
5.      Label l;
6.      Color c=Color.BLUE;
7.      Paint(){
8.      l=new Label();
9.      l.setBounds(20,40,100,20);
10.     add(l);
11.
12.     addMouseMotionListener(this);
13.
14.     setSize(400,400);
15.     setLayout(null);
16.     setVisible(true);
17. }
18. public void mouseDragged(MouseEvent e) {
19.     l.setText("X="+e.getX()+", Y="+e.getY());
20.     Graphics g=getGraphics();
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
21.     g.setColor(Color.RED);
22.     g.fillOval(e.getX(),e.getY(),20,20);
23. }
24. public void mouseMoved(MouseEvent e) {
25.     l.setText("X="+e.getX()+", Y="+e.getY());
26. }
27. public static void main(String[] args) {
28.     new Paint();
29. }
30. }
```

Output:



Java ItemListener Interface

The Java ItemListener is notified whenever you click on the checkbox. It is notified against ItemEvent. The ItemListener interface is found in java.awt.event package. It has only one method: itemStateChanged().

itemStateChanged() method

The itemStateChanged() method is invoked automatically whenever you click or unclick on the registered checkbox component.

```
1. public abstract void itemStateChanged(ItemEvent e);
```

Java ItemListener Example

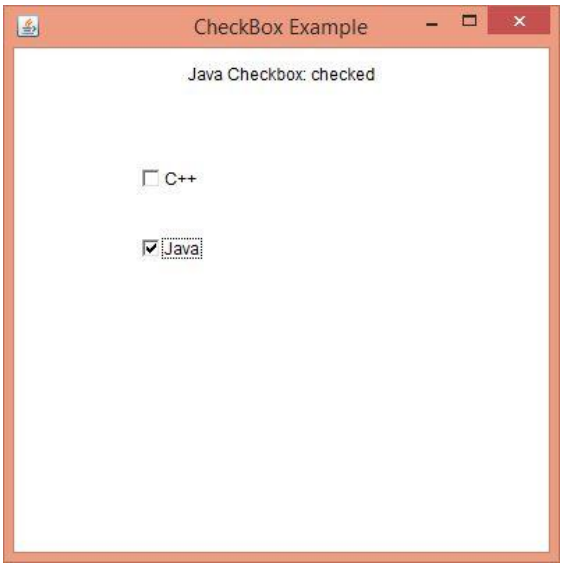
```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class ItemListenerExample implements ItemListener{
4.     Checkbox checkBox1,checkBox2;
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
5.      Label label;
6.      ItemListenerExample(){
7.          Frame f= new Frame("CheckBox Example");
8.          label = new Label();
9.          label.setAlignment(Label.CENTER);
10.         label.setSize(400,100);
11.         checkBox1 = new Checkbox("C++");
12.         checkBox1.setBounds(100,100, 50,50);
13.         checkBox2 = new Checkbox("Java");
14.         checkBox2.setBounds(100,150, 50,50);
15.         f.add(checkBox1); f.add(checkBox2); f.add(label);
16.         checkBox1.addItemListener(this);
17.         checkBox2.addItemListener(this);
18.         f.setSize(400,400);
19.         f.setLayout(null);
20.         f.setVisible(true);
21.     }
22.    public void itemStateChanged(ItemEvent e) {
23.         if(e.getSource()==checkBox1)
24.             label.setText("C++ Checkbox: "
25.             + (e.getStateChange()==1?"checked":"unchecked"));
26.         if(e.getSource()==checkBox2)
27.         label.setText("Java Checkbox: "
28.         + (e.getStateChange()==1?"checked":"unchecked"));
29.     }
30. public static void main(String args[])
31. {
32.     new ItemListenerExample();
33. }
34. }
```

Output:

Reference:- www.javatpoint.com and www.tutorialspoint.com



Java KeyListener Interface

The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package. It has three methods.

Methods of KeyListener interface

The signature of 3 methods found in KeyListener interface are given below:

```
1. public abstract void keyPressed(KeyEvent e);
2. public abstract void keyReleased(KeyEvent e);
3. public abstract void keyTyped(KeyEvent e);
```

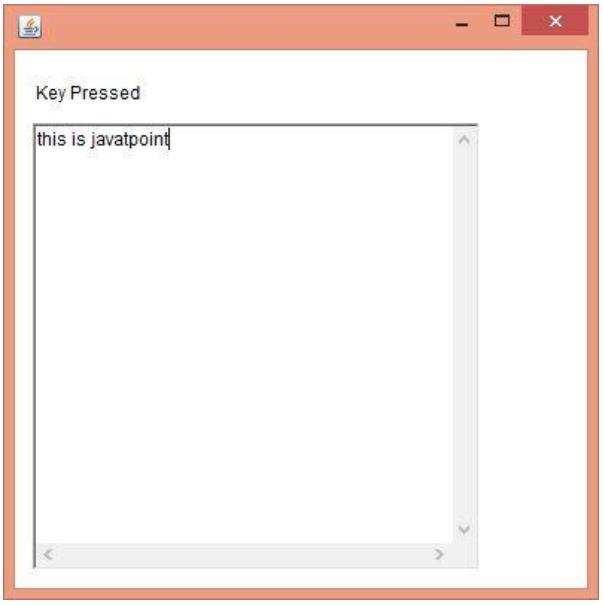
Java KeyListener Example

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class KeyListenerExample extends Frame implements KeyListener{
4.     Label l;
5.     TextArea area;
6.     KeyListenerExample(){
7.
8.         l=new Label();
9.         l.setBounds(20,50,100,20);
10.        area=new TextArea();
11.        area.setBounds(20,80,300, 300);
12.        area.addKeyListener(this);
13.
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
14.         add(l);add(area);
15.         setSize(400,400);
16.         setLayout(null);
17.         setVisible(true);
18.     }
19.     public void keyPressed(KeyEvent e) {
20.         l.setText("Key Pressed");
21.     }
22.     public void keyReleased(KeyEvent e) {
23.         l.setText("Key Released");
24.     }
25.     public void keyTyped(KeyEvent e) {
26.         l.setText("Key Typed");
27.     }
28.
29.     public static void main(String[] args) {
30.         new KeyListenerExample();
31.     }
32. }
```

Output:



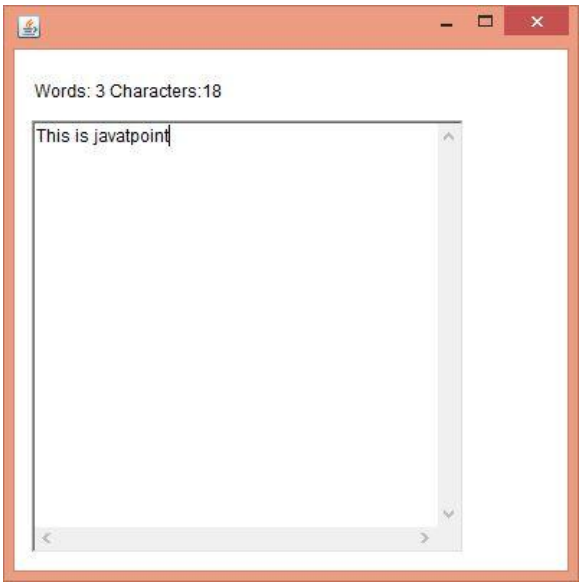
Java KeyListener Example 2: Count Words & Characters

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class KeyListenerExample extends Frame implements KeyListener{
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
4.      Label l;
5.      TextArea area;
6.      KeyListenerExample(){
7.
8.          l=new Label();
9.          l.setBounds(20,50,200,20);
10.         area=new TextArea();
11.         area.setBounds(20,80,300, 300);
12.         area.addKeyListener(this);
13.
14.         add(l);add(area);
15.         setSize(400,400);
16.         setLayout(null);
17.         setVisible(true);
18.     }
19.     public void keyPressed(KeyEvent e) {}
20.     public void keyReleased(KeyEvent e) {
21.         String text=area.getText();
22.         String words[]=text.split("\\s");
23.         l.setText("Words: "+words.length+" Characters:"+text.length());
24.     }
25.     public void keyTyped(KeyEvent e) {}
26.
27.     public static void main(String[] args) {
28.         new KeyListenerExample();
29.     }
30. }
```

Output:



Java WindowListener Interface

The Java WindowListener is notified whenever you change the state of window. It is notified against WindowEvent. The WindowListener interface is found in java.awt.event package. It has three methods.

Methods of WindowListener interface

The signature of 7 methods found in WindowListener interface are given below:

1. **public abstract void** windowActivated(WindowEvent e);
2. **public abstract void** windowClosed(WindowEvent e);
3. **public abstract void** windowClosing(WindowEvent e);
4. **public abstract void** windowDeactivated(WindowEvent e);
5. **public abstract void** windowDeiconified(WindowEvent e);
6. **public abstract void** windowIconified(WindowEvent e);
7. **public abstract void** windowOpened(WindowEvent e);

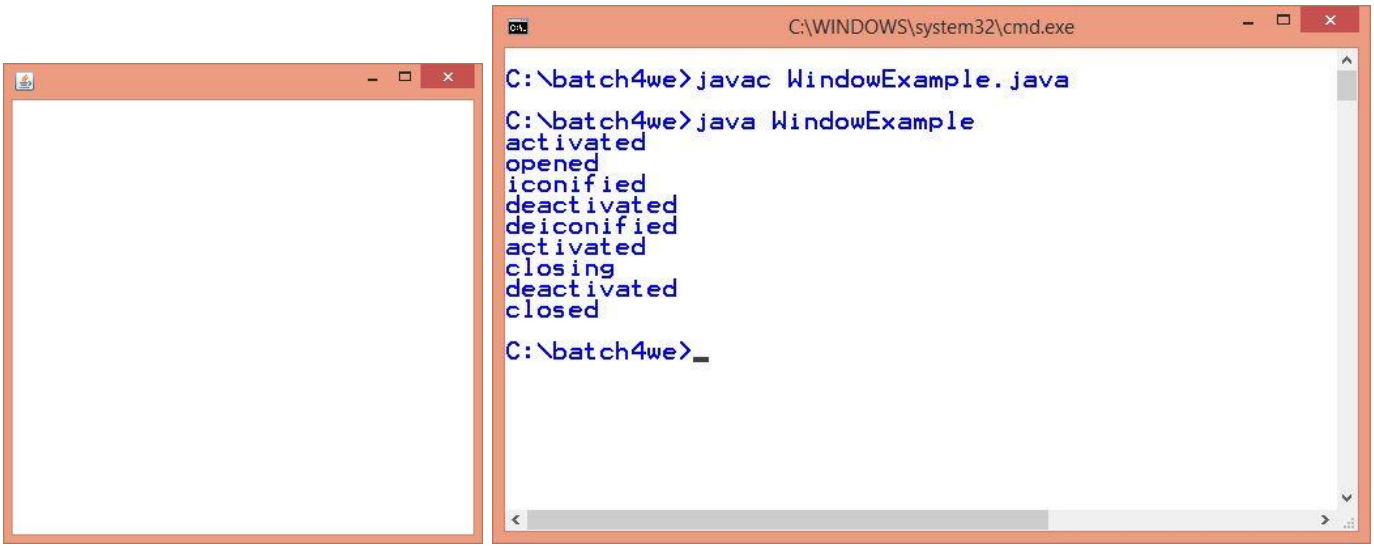
Java WindowListener Example

```
1. import java.awt.*;
2. import java.awt.event.WindowEvent;
3. import java.awt.event.WindowListener;
4. public class WindowExample extends Frame implements WindowListener{
5.     WindowExample(){
6.         addWindowListener(this);
7.
8.         setSize(400,400);
9.         setLayout(null);
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
10.         setVisible(true);
11.     }
12.
13. public static void main(String[] args) {
14.     new WindowExample();
15. }
16. public void windowActivated(WindowEvent arg0) {
17.     System.out.println("activated");
18. }
19. public void windowClosed(WindowEvent arg0) {
20.     System.out.println("closed");
21. }
22. public void windowClosing(WindowEvent arg0) {
23.     System.out.println("closing");
24.     dispose();
25. }
26. public void windowDeactivated(WindowEvent arg0) {
27.     System.out.println("deactivated");
28. }
29. public void windowDeiconified(WindowEvent arg0) {
30.     System.out.println("deiconified");
31. }
32. public void windowIconified(WindowEvent arg0) {
33.     System.out.println("iconified");
34. }
35. public void windowOpened(WindowEvent arg0) {
36.     System.out.println("opened");
37. }
38. }
```

Output:



Java Adapter Classes

Java adapter classes *provide the default implementation of listener interfaces*. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it *saves code*.

The adapter classes are found in **java.awt.event**, **java.awt.dnd** and **javax.swing.event** packages. The Adapter classes with their corresponding listener interfaces are given below.

java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener

Reference:- www.javatpoint.com and www.tutorialspoint.com

HierarchyBoundsAdapter	HierarchyBoundsListener

java.awt.dnd Adapter classes

Adapter class	Listener interface
DragSourceAdapter	DragSourceListener
DragTargetAdapter	DragTargetListener

javax.swing.event Adapter classes

Adapter class	Listener interface
MouseInputAdapter	MouseInputListener
InternalFrameAdapter	InternalFrameListener

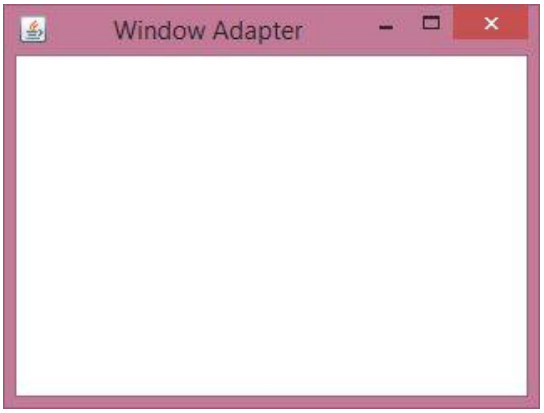
Java WindowAdapter Example

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class AdapterExample{
4.     Frame f;
5.     AdapterExample(){
6.         f=new Frame("Window Adapter");
7.         f.addWindowListener(new WindowAdapter(){
8.             public void windowClosing(WindowEvent e) {
9.                 f.dispose();
10.             }
11.         });
12.
13.         f.setSize(400,400);
14.         f.setLayout(null);
15.         f.setVisible(true);
16.     }
17. public static void main(String[] args) {
18.     new AdapterExample();
19. }
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
20. }
```

Output:

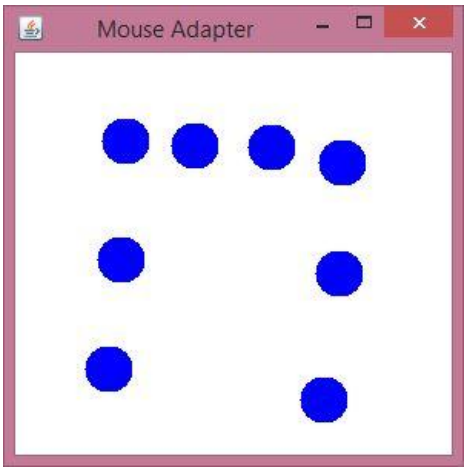


Java MouseAdapter Example

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  public class MouseAdapterExample extends MouseAdapter{
4.     Frame f;
5.     MouseAdapterExample(){
6.        f=new Frame("Mouse Adapter");
7.        f.addMouseListener(this);
8.
9.        f.setSize(300,300);
10.       f.setLayout(null);
11.       f.setVisible(true);
12.    }
13.    public void mouseClicked(MouseEvent e) {
14.       Graphics g=f.getGraphics();
15.       g.setColor(Color.BLUE);
16.       g.fillOval(e.getX(),e.getY(),30,30);
17.    }
18.
19. public static void main(String[] args) {
20.    new MouseAdapterExample();
21. }
22. }
```

Output:

Reference:- www.javatpoint.com and www.tutorialspoint.com

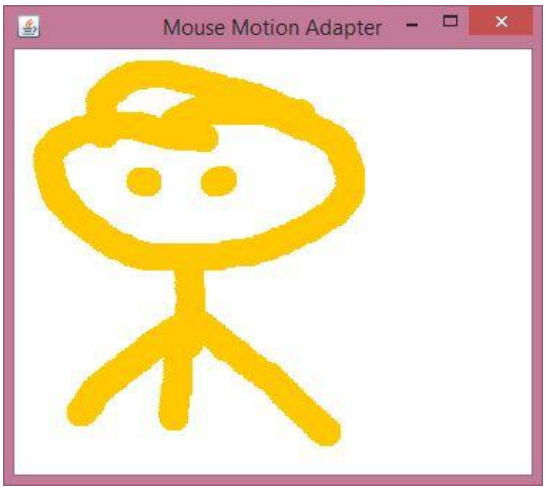


Java MouseMotionAdapter Example

```
1. import java.awt.*;
2. import java.awt.event.*;
3. public class MouseMotionAdapterExample extends MouseMotionAdapter{
4.     Frame f;
5.     MouseMotionAdapterExample(){
6.         f=new Frame("Mouse Motion Adapter");
7.         f.addMouseMotionListener(this);
8.
9.         f.setSize(300,300);
10.        f.setLayout(null);
11.        f.setVisible(true);
12.    }
13. public void mouseDragged(MouseEvent e) {
14.    Graphics g=f.getGraphics();
15.    g.setColor(Color.ORANGE);
16.    g.fillOval(e.getX(),e.getY(),20,20);
17. }
18. public static void main(String[] args) {
19.    new MouseMotionAdapterExample();
20. }
21. }
```

Output:

Reference:- www.javatpoint.com and www.tutorialspoint.com



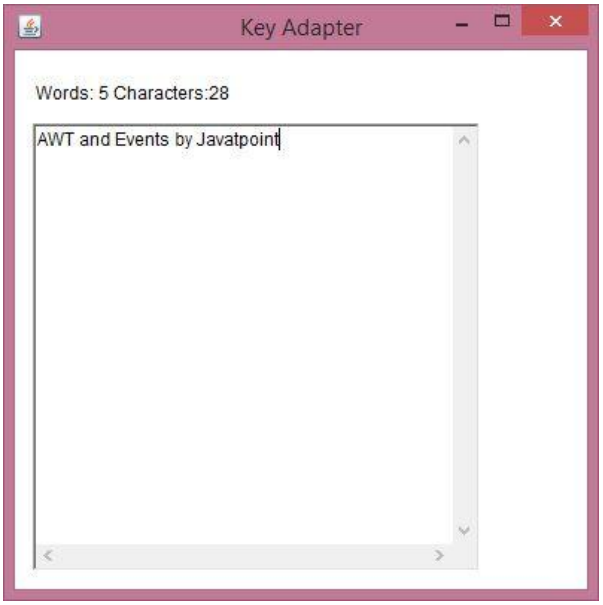
Java KeyAdapter Example

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  public class KeyAdapterExample extends KeyAdapter{
4.      Label l;
5.      TextArea area;
6.      Frame f;
7.      KeyAdapterExample(){
8.          f=new Frame("Key Adapter");
9.          l=new Label();
10.         l.setBounds(20,50,200,20);
11.         area=new TextArea();
12.         area.setBounds(20,80,300, 300);
13.         area.addKeyListener(this);
14.
15.         f.add(l);f.add(area);
16.         f.setSize(400,400);
17.         f.setLayout(null);
18.         f.setVisible(true);
19.     }
20.     public void keyReleased(KeyEvent e) {
21.         String text=area.getText();
22.         String words[]=text.split("\\s");
23.         l.setText("Words: "+words.length+" Characters:"+text.length());
24.     }
25.
26.     public static void main(String[] args) {
27.         new KeyAdapterExample();
28.     }
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

29. }

Output:



How to close AWT Window in Java

We can close the AWT Window or Frame by calling *dispose()* or *System.exit()* inside windowClosing() method. The windowClosing() method is found in **WindowListener** interface and **WindowAdapter** class.

The WindowAdapter class implements WindowListener interfaces. It provides the default implementation of all the 7 methods of WindowListener interface. To override the windowClosing() method, you can either use WindowAdapter class or WindowListener interface.

If you implement the WindowListener interface, you will be forced to override all the 7 methods of WindowListener interface. So it is better to use WindowAdapter class.

Different ways to override windowClosing() method

There are many ways to override windowClosing() method:

- By anonymous class
- By inheriting WindowAdapter class
- By implementing WindowListener interface

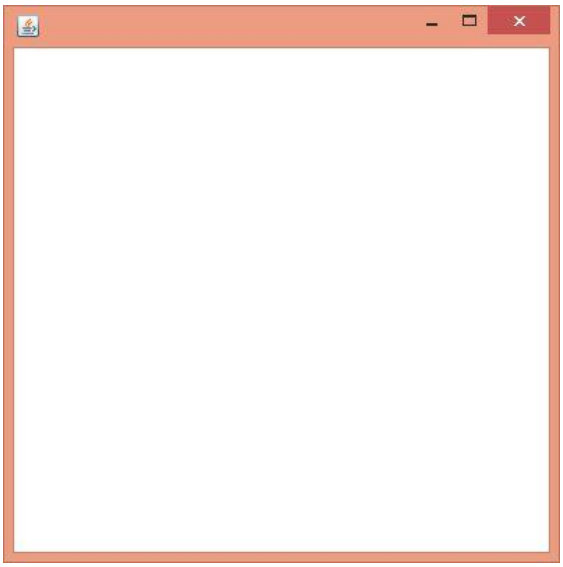
Close AWT Window Example 1: Anonymous class

```
1. import java.awt.*;
2. import java.awt.event.WindowEvent;
3. import java.awt.event.WindowListener;
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
4.  public class WindowExample extends Frame{
5.      WindowExample(){
6.          addWindowListener(new WindowAdapter(){
7.              public void windowClosing(WindowEvent e) {
8.                  dispose();
9.              }
10.         });
11.         setSize(400,400);
12.         setLayout(null);
13.         setVisible(true);
14.     }
15. public static void main(String[] args) {
16.     new WindowExample();
17. }
```

Output:



Close AWT Window Example 2: extending WindowAdapter

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.  public class AdapterExample extends WindowAdapter{
4.      Frame f;
5.      AdapterExample(){
6.          f=new Frame();
7.          f.addWindowListener(this);
8.
9.          f.setSize(400,400);
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

```
10.         f.setLayout(null);
11.         f.setVisible(true);
12.     }
13. public void windowClosing(WindowEvent e) {
14.     f.dispose();
15. }
16. public static void main(String[] args) {
17.     new AdapterExample();
18. }
19. }
```

Close AWT Window Example 3: implementing WindowListener

```
1.  import java.awt.*;
2.  import java.awt.event.WindowEvent;
3.  import java.awt.event.WindowListener;
4.  public class WindowExample extends Frame implements WindowListener{
5.      WindowExample(){
6.          addWindowListener(this);
7.
8.          setSize(400,400);
9.          setLayout(null);
10.         setVisible(true);
11.     }
12.
13. public static void main(String[] args) {
14.     new WindowExample();
15. }
16. public void windowActivated(WindowEvent e) {}
17. public void windowClosed(WindowEvent e) {}
18. public void windowClosing(WindowEvent e) {
19.     dispose();
20. }
21. public void windowDeactivated(WindowEvent e) {}
22. public void windowDeiconified(WindowEvent e) {}
23. public void windowIconified(WindowEvent e) {}
24. public void windowOpened(WindowEvent arg0) {}
25. }
```

Reference:- www.javatpoint.com and www.tutorialspoint.com

MODULE - V

Java Database Connectivity (JDBC)

MODULE 5

Java Database Connectivity (JDBC)

Module Description

Java Database Connectivity (JDBC) is an Application Programming Interface (API) used to connect Java application with Database. JDBC is used to interact with the various type of Database such as Oracle, MS Access, My SQL and SQL Server and it can be stated as the platform-independent interface between a relational database and Java programming.

This JDBC chapter is designed for Java programmers who would like to understand the JDBC framework in detail along with its architecture and actual usage.

iNurture's **Programming in Java** course is designed to serve as a stepping stone for you to build a career in Information Technology.

Chapter Table of Contents

Chapter 5.1

Introduction to JDBC

Aim

To provide students with a basics concepts of JDBC

Instructional Objectives

After completing this chapter, you should be able to:

- Explain database connectivity
- Demonstrate JDBC architecture*
- Explain different types of JDBC drivers
- Illustrate JDBC API
- Explain the basic steps to create a JDBC application

Learning Outcomes

At the end of this chapter, you are expected to:

- Describe JDBC architecture
- Identify the types of JDBC driver
- Describe the types of statements in JDBC
- Differentiate prepared statement and statement
- Develop a java program and show the database connectivity using JDBC

5.1.1 Database Connectivity

Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is part of the Java Standard Edition platform, from Oracle Corporation. It provided methods to query and update data in a database and related towards relational databases. A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual machine (JVM) host environment.

The JDBC library combines APIs for each of the tasks discussed below that are associated with database usage.

1. Create a connection to a database.
2. Creating SQL or MySQL statements.
3. Executing SQL or MySQL queries in the database.
4. Viewing & modifying the resulting records.

JDBC is a specification that provides a comprehensive set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executable, such as:

1. Java Applications
2. Java Applets
3. Java Servlets
4. Java Server Pages (JSPs)
5. Enterprise JavaBeans (EJBs).

All of these different executable can use a JDBC driver to access a database and take advantage of the stored data. JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code. JDBC connection supports to create and execute statements. These statements are an updated statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT.

Statement are termed as a report which is sent to the database server all time. Prepared Statements are statements which are cached and the execution path is pre-determined on the database server. Callable Statements are used for executing stored methods.

Update statements such as INSERT, UPDATE and DELETE return indicates how many rows were affected in the database.

These statements rather do not return information.

Query statements come to a JDBC row result set. Separate columns in a row are retrieved by name or by its number. It may have any number of rows in the result set.

Using JDBC

1. **To execute a statement against a database, the following flow is observed**
   - Load the driver (Only performed once)
   - Obtain a Connection to the database (Save for later use)
   - Get a Statement object from the Connection
   - Use the Statement object to execute SQL. Updates, inserts and deletes return Boolean. Selects return a ResultSet
   - Navigate ResultSet, using data as required
   - Close ResultSet
   - Close Statement
2. **DO NOT close the connection**
   - The same connection object can be used to create further statements.
   - A Connection may only have one active Statement at a time. Do not forget to close the statement when it is no longer needed.
   - Close the connection when you no longer need to access the database.
3. **Using a connection**

The Connection interface defines many methods for managing and using a connection to the database. Few methods are:

public Statement createStatement( )

public PreparedStatement prepareStatement(String sql)

public void setAutoCommit(boolean)

public void commit( )

public void rollback( )

public void close( )

The most commonly used method is createStatement( ).These statements are an updated statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT. Statements are termed as a report which is sent to the database server all time. Prepared Statements are statements which are cached and the execution path is pre-determined on the database server. When an SQL statement issues against the database, a Statement object must be created through the Connection.

Types of Connections are made to Database Servers

1. Direct connections

Having a direct relationship means you directly connect from the client to the database without an average service. To get a direct connection to a database, a programmer must have the correct direct connect drivers installed on the connecting client.

When attached to a database server and the databases stored on it, you must use a Windows-authenticated login. It is a process of identifying an individual user with credentials supplied by the Windows operating system of the user's computer. Hence, the login with which you log in to the client computer is the login that uses the connection. When connecting to a remote database server make use of a domain login. When connecting to a local database server, use either a local or a domain login. If you use a domain login while connecting to a local database server, you may not be able to log into the database server.

2. Local vs. remote connections

When connecting to database servers which reside on the same computer as the connecting client application, make use of a local or domain account to log in.

For example, if your local login is mymachine\myuser, creating a login with the same name on the remote machine, your computer, results in this login: yourmachine\myuser. These are essentially two different login names.

Self-assessment Questions

1) Identify which of the following is NOT a component/class of JDBC API?
   a) Statement
   b) ResultSet
   c) SQLException
   d) ConnectionPool

2) Choose one of the following steps to establish a connection with a database?
   a) Import packages containing the JDBC classes needed for database programming.
   b) Register the JDBC driver, so that you can open a communications channel with the database.
   c) Open a connection using the DriverManager. getConnection( ) method.
   d) Execute a query using an object of type Statement.

3) Methods for contacting a database is Connection.
   a) True
   b) False

4) JDBC architecture decouples an abstraction from its implementation and follows a bridge design pattern.
   a) True
   b) False

5) Is there any possibility for implementing an interface in JSP?
   a) Yes
   b) No
   c) None
   d) Can't say

5.1.2 JDBC Architecture

The JDBC API support both two-tier and three-tier processing models for database access but in usually, JDBC Architecture consists of two layers:

1. **JDBC Driver API:** JDBC Driver API supports the JDBC Manager-to-Driver Connection.

2. **JDBC API:** This provides the application-to-JDBC Manager connection.

The JDBC API applies a database-specific drivers and driver manager to provide good connectivity to create heterogeneous databases.

The JDBC driver manager secures that the correct driver is used to obtain each data source. The driver manager is able to support multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the JDBC architectural diagram, which shows the location of the driver manager on the JDBC drivers and the Java application:

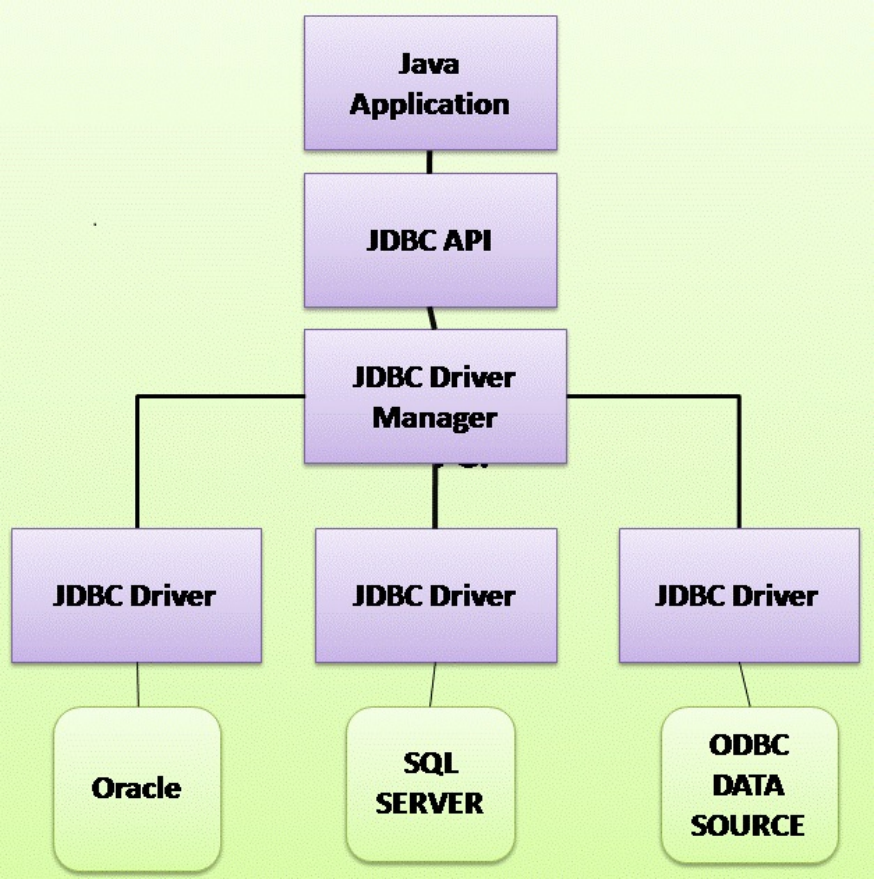


Figure 5.1.1: JDBC Architecture

Common JDBC Component:

These are the following interfaces and classes provided by JDBC API:

- **Driver Manager:** This class manages a list of database drivers. Matches connection requests from the Java application with the proper database driver using communication sub protocol. The first driver that recognises a certain sub protocol under JDBC will be used to establish a database Connection.

- **Driver:** This interface handles the connections with the database server. You will interact directly with Driver objects very rarely. Instead, JDBC uses Driver Manager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, *i.e.,* all communication with the database is through connection object only.

- **Statement:** You use objects created to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

- **SQLException:** This class handles any errors that occur in a database application.

Different types of JDBC drivers are explained in the following topics and the statements of JDBC is defined with examples. The following are the steps to create a JDBC application.

The necessary steps to create a JDBC application.

There are following six steps involved in building a JDBC application:

Step 1: Import the packages: Needs that coder include the packages including the JDBC classes needed for database programming. Often, we will use "using import java.sql.*;" as suffice.

Step 2: Register the JDBC driver: Requires that you initialise a driver so you can open a communication channel with the database.

Step 3: Open a connection: Requires using the DriverManager.getConnection( ) method to create a Connection object, which represents a physical connection with the database.

Step 4: Execute a query: Requires using an object of type Statement for building and submitting an SQL statement to the database.

Step 5: Extract data from result set: Requires that you use the appropriateResultSet.get( ) method to retrieve the data from the result set.

Step 6: Clean up the environment: Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

For example,

```
//Step 1:   Import required packages
import java.sql.*;
public class JdbcCon
{
   // JDBC driver name and database URL
   static final String JDBC_DRIVER = "com.sql.jdbc.Driver";
   static final String DB_URL = "jdbc:sql://localhost/EMPE";
   // Database credentials
   static final String UNAME = "username";
   static final String UPASS = "password";
   public static void main(String[] args)
   {
      Connection con = null;
      Statement stmtt = null;
      try
      {
         //Step 2: Register JDBC driver
         Class.forName("com.mysql.jdbc.Driver");
         //Step 3:   Open a connection
         System.out.println("Connecting to database...");
         con = DriverManager.getConnection(DB_URL,UNAME ,UPASS);
         //Step 4:   Execute a query
         System.out.println("Creating statement...");
         stmtt = conn.createStatement();
         String mysql;
         sql = "SELECT id, firstname, lastname, age FROM student";
         ResultSet rts = stmtt.executeQuery(sql);
         //Step 5:  Extract data from result set
         while(rts.next())
         {
            //Retrieve by column name
            int id = rts.getInt("id");
            int age = rts.getInt("yage");
            String first = rts.getString("first");
```

```
            String last = rts.getString("last");
            //Display values
            System.out.print("ID: " + id);
            System.out.print(", YourAge: " + age);
            System.out.print(", First: " + firstname);
            System.out.println(", Last: " + lastname);
         }
         //Step 6:   Clean-up environment
         rts.close();
         stmtt.close();
         con.close();
      }
      catch(SQLException se)
      {
         //Handle errors for JDBC
         se.printStackTrace();
      }
      catch(Exception e)
      {
         //Handle errors for Class.forName
         e.printStackTrace();
      }
      Finally
      {
         //finally block used to close resources
         try
         {
            if(stmtt!=null)
            stmtt.close();
         }
         catch(SQLException se2)
         { }// nothing we can do
         try{
         if(con!=null)
         con.close();
      }
      catch(SQLException se)
      {
         se.printStackTrace();
      }
      //end finally try
      }
      //end try
      System.out.println("Goodbye!");
   }
   //end main
}
//end FirstExample
```

Now we are going to compile the above example as follows:

C:\>javac FirstExample.java

C:\>

When you run FirstExample, it produces the following result:

C:\>java FirstExample

Connecting to database...

Creating statement...

ID: 10, YourAge: 18, First: Zara, Last: Khan

ID: 11, YourAge : 25, First: Prabha, Last: karan

ID: 12, YourAge: 30, First: Zaid, Last: Ali

ID: 13, YourAge: 28, First: Sunil, Last: Mittal

C:\>

Self-assessment Questions

6) How many layers JDBC architecture consists of?

a) 1 b) 3

c) 2 d) 5

7) JDBC API support_____ processing models for database access.

a) Only two-tier b) Only three-tier

c) Both a and b

5.1.3 JDBC Drivers

JDBC driver is used for interacting with the database server which implements the stated interfaces in the JDBC API.

For example, JDBC drivers interact with the database to open database connections by sending SQL or database commands and receive the results with Java:

- Any proprietary APIs are implemented by a JDBC driver.

Types of JDBC Drivers

There are **four** different types of JDBC drivers:

1. **Type 1:** JDBC - ODBC bridge driver
2. **Type 2:** Java - Native code driver
3. **Type 3:** Network - Protocol driver (Middleware driver)
4. **Type 4:** Database - Protocol driver (Pure Java driver)

(i) JDBC – ODBC Bridge

ODBC driver is a type 1 driver installed on every client machine. It can be accessible by the JDBC Bridge which supports ODBC accessibility but at present this type is recommended only for experimental purposes.

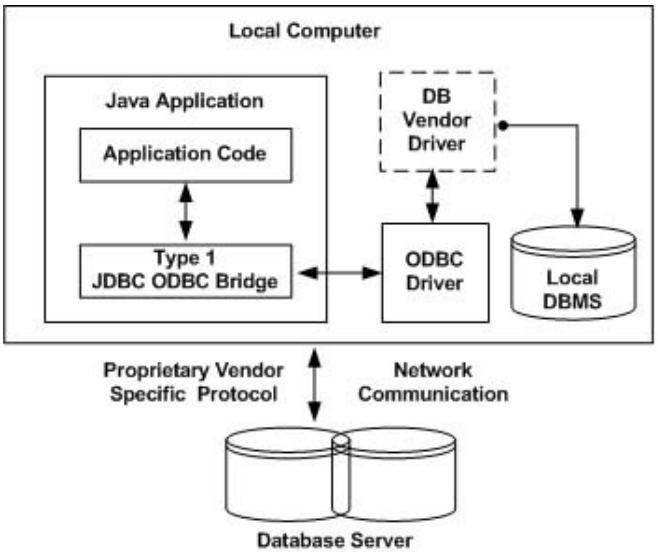


Figure 5.1.2: JDBC- ODBC Bridge

In a Type 1 driver (JDBC – ODBC Bridge), a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring your system a Data Source Name (DSN) that represents the mark database. When Java first came out, JDBC – ODBC Bridge was a valuable driver because most databases only supported ODBC access but now this type of driver is approved only for experimental use or when no another alternative is available.

The JDBC-ODBC Bridge that comes with JDBC version JDK 1.2 is a good example of this kind of driver.

Advantages:

- Easy to use.
- Can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC calls converts into ODBC function calls.
- The ODBC driver requires installation on the client machine.

(ii) Native – API Driver

JDBC Native API is unique to the database because these API calls are converted into native C/C++ API calls. They are provided by the database vendors and used in a similar manner as the JDBC-ODBC Bridge.

The vendor-specific driver must be installed on each client machine and if the programmer wants to change the database, then the programmer has to modify the native API too.

JDBC-Native API can perform faster when compared to Type 1 driver because it eliminates ODBC's Overhead.

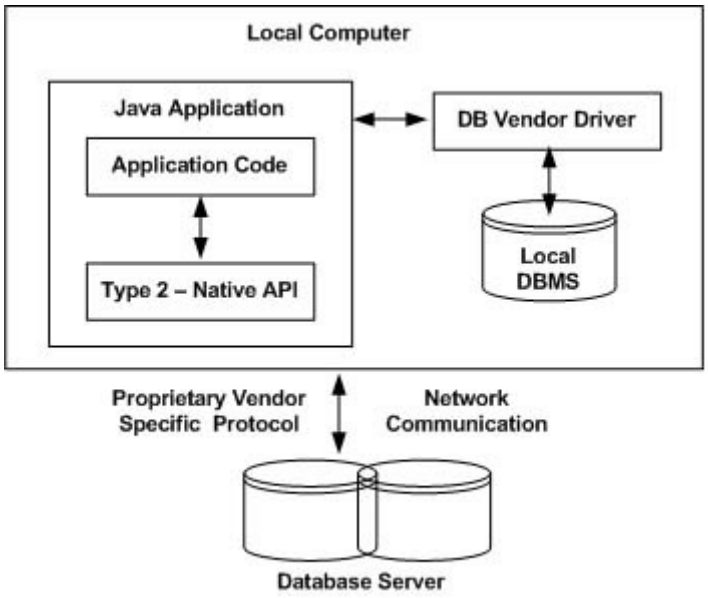


Figure 5.1.3: Native API Driver

The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Advantage:

- Performance upgrades than JDBC-ODBC bridge driver.

Disadvantages:

- The Native driver requires installation on the each client machine.
- The Vendor client library requires installation on client machine.

(iii) Network – Protocol Driver (Middleware Driver)

Network-Protocol driver is also called as middleware driver. It has a three-tier approach to access databases. It communicates with the middleware application server through a standard network socket. Then the device information is translated by the middleware application server as a call format required by the DBMS and forwarded to the database server.

Middleware drivers are extremely flexible since no code is installed on the client side. Even a single driver can provide access to multiple databases.

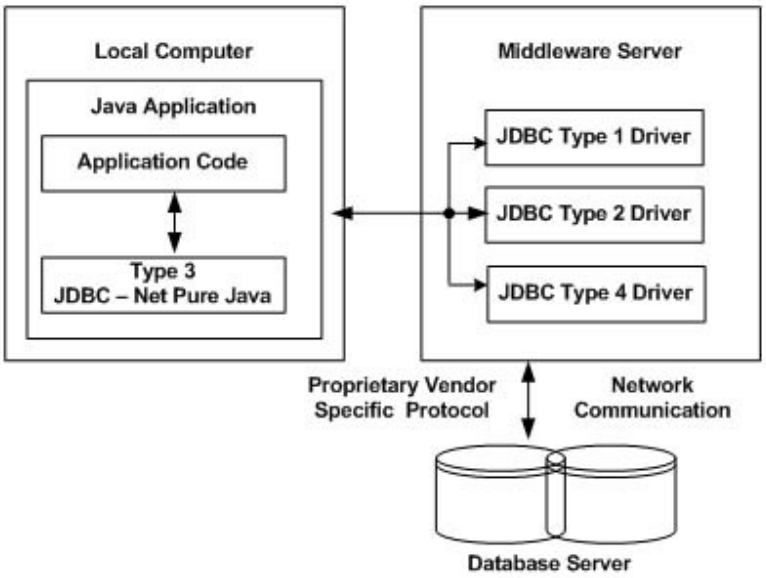


Figure 5.1.4: Middleware Driver

You can analyse the application server as a JDBC "proxy," meaning that it creates calls to the client application. As a result, you need some information about the application server's configuration to effectively use this driver type. Your application server might use a Type 1, 2, or 4 drivers to communicate with the database, understanding the nuances will prove helpful.

Advantage:

- No client-side library needs the application server that can perform many tasks like auditing, load balancing, logging, etc.

Disadvantages:

- Network support is necessary on the client machine.
- It requires database-specific coding for the middle tier.
- Maintenance of Network Protocol driver is costly because it requires database-specific coding in the middle tier.

(iv) Database – Protocol Driver (Pure Java Driver)

Database- Protocol driver is a type 4 driver called as a pure Java driver. It communicates directly with the vendor's database through socket information. It acts as the highest performance driver available for the database which is provided by the supplier itself.

Pure Java driver does not require any special software to install on the client or server side rather these drivers can be downloaded dynamically.

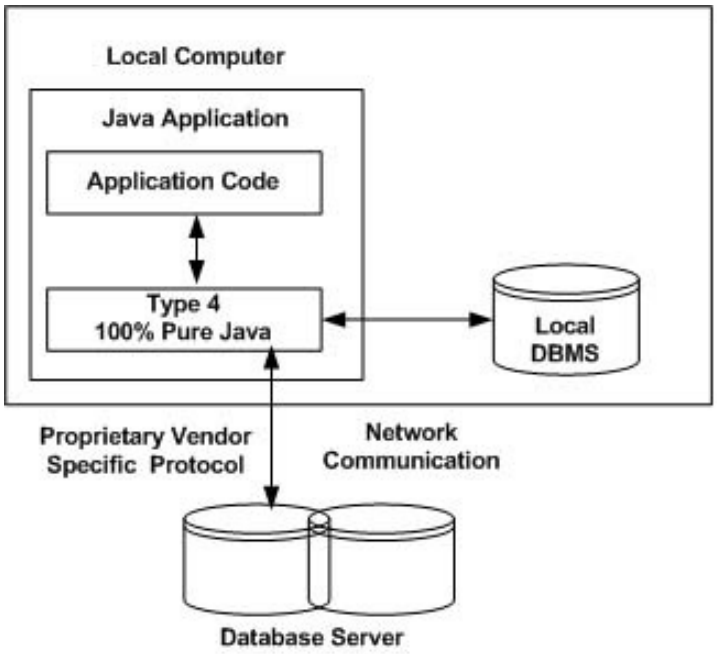


Figure 5.1.5: 100% Pure Java Driver

This driver named as MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors supply type 4 drivers.

Advantages:

- Better performance than all other drivers.
- No software is required on the client side or server side.

Disadvantage:

- Drivers depends on the Database.

Self-assessment Questions

7) Which is the type of JDBC driver, calls native code of the locally available ODBC driver?
   a) JDBC-ODBC Bridge plus ODBC driver
   b) Native-API, partly Java driver
   c) JDBC-Net, pure Java driver
   d) Native-protocol, pure Java driver

8) Which of the following type of JDBC driver, is also called Type 3 JDBC driver?
   a) JDBC-ODBC Bridge plus ODBC driver
   b) Nativ e-API, partly Java driver
   c) JDBC-Net, pure Java driver
   d) Native-protocol, pure Java driver

9) A native-protocol driver allows a direct call from the client machine to the DBMS server.
   a) True b) False

10) JDBC - ODBC Bridge provides JDBC API access via ODBC drivers.
   a) True b) False

5.1.4 JDBC API

Java programming language provides a universal data access to the Java Database Connectivity (JDBC) API. The application of JDBC API is that you can access virtually any data source, relational databases to spreadsheets and files. It provides a common base on which tools and alternate interfaces evolve.

The JDBC API has two packages:

1. java.sql
2. javax.sql

Java Database Connectivity (JDBC) API:

- General API (Application programming interface) for Java client code to connect to a database
- Database providers (IBM, Oracle, etc.) provide drivers
- **Driver:** specific implementation of the API for interacting with a particular database
- Support for
    a) Statement
    b) Prepared Statement
    c) Callable Statement (stored procedures)
    d) Common Java data types (Integer, Double, java.sql.Date)
- The javax.sql and java.sql are the primary packages for JDBC 4.0. It is the modern JDBC version at the time of writing this tutorial. It offers the leading (main) classes for interacting with your data sources. The new features in these packages include changes in the following areas:
    - Automatic database driver loading.
    - Exception handling improvements.
    - Enhanced BLOB/CLOB functionality.
    - Connection and statement interface enhancements.
    - National character set support.
    - SQL ROWID access.

- SQL 2003 XML data type support.
- Annotations.

Self-assessment Questions

11) The application of JDBC API is that you cannot access virtually any data source, relational databases to spreadsheets and files.

a) True b) False

12) The abbreviation of API

a) Application programming interface
b) Anti-programming interface
c) Abdicate programming interface
d) Abscising programming interface

13) Which processing models supports database access?

a) Only two-tier b) Only three-tier
c) Both two-tier and three-tier d) None of these

14) The latest version of JDBC 4.0 API is divided into two packages, these are?

1. java.sql
2. javax.sql
3. javac.sql
4. java.lang.sql

a) 2 and 3 b) 1 and 2
c) 1 and 3 d) 1 and 4

15) Select the API that provides the application-to-JDBC Manager connection?

a) JDBC API b) JDBC Driver API
c) Both JDBC API JDBC Driver API d) None of these

16) The JDBC API provide facility the facility for accessing the _____ from the java programming language.

a) Database b) Relational database
c) Irrational database d) None of these

Summary

- Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database.
- It is part of the Java Standard Edition platform, from Oracle Corporation. It provided methods to query and update data in a database and related towards relational databases.
- A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual machine (JVM) host environment.
- In Java database connectivity JDBC API is a Java API that can access any tabular data, especially data stored in a Relational Database.
- JDBC works with Java on a variety of platforms, such as Windows, Mac OS and the various versions of UNIX.
- JDBC driver is used for interacting with the database server which implements the stated interfaces in the JDBC API.
- The JDBC driver manager secures that the correct driver is used to obtain each data source. The driver manager is able to support multiple concurrent drivers connected to multiple heterogeneous databases
- Having a direct relationship means you directly connect from the client to the database without an average service. To get a direct connection to a database, a programmer must have the correct direct connect drivers installed on the connecting client.

Terminal Questions

1. Explain database connectivity
2. Demonstrate JDBC architecture
3. Explain different types of JDBC drivers
4. Illustrate JDBC API
5. Explain the basic steps to create a JDBC application

Answer Keys

Self-assessment Questions	
Question No.	Answer
1	d
2	c
3	a
4	a
5	b
6	c
7	a
8	c
9	a
10	b
11	b
12	a
13	c
14	b
15	b
16	b

Activity

Activity Type: Offline **Duration:** 30 Minutes

Description:

Compare Type 1, Type 2, Type 3 and Type 4 JDBC drivers. Create a table which describe the features, advantages and disadvantages of these JDBC drivers.

Bibliography

External Resources

- The complete reference Java-2: Herbert Schildt-7th edition- McGraw Hill professional
- SAMS teach yourself Java-2- Rogers Cedenhead and Leura Lemay- 3rd edition – Pearson
- JDBC and Java database programming books, Laurence Vanhelsuwe

Video Links

Topic	Link
Architecture	https://docs.oracle.com/javase/tutorial/jdbc/overview/architecture.html
Types of Drive	http://tutorials.jenkov.com/jdbc/driver-types.html
Drive	https://www.youtube.com/watch?v=bo62AqckwHc
Types of Drive	http://tutorials.jenkov.com/jdbc/driver-types.html

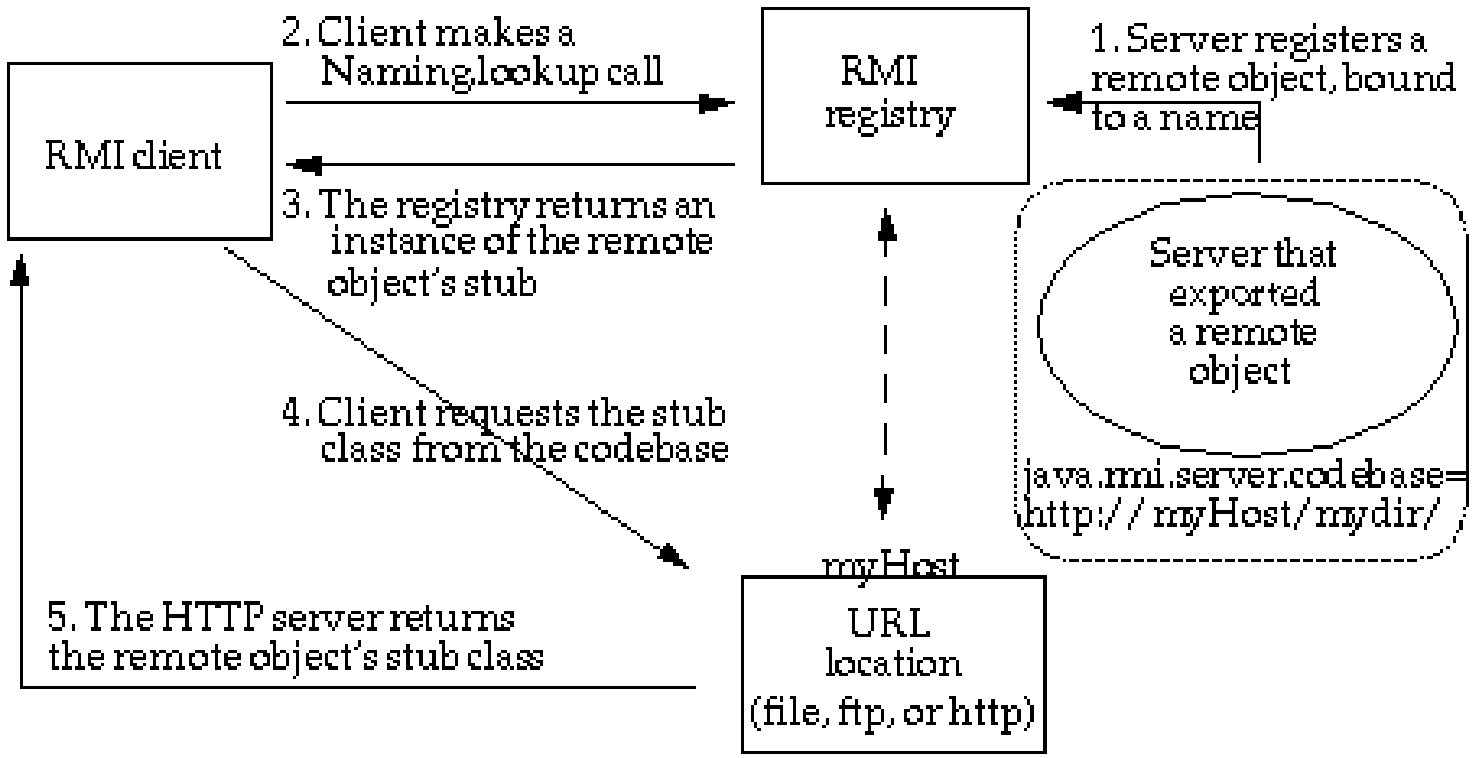
Notes:

Java RMI
(Remote Method Invocation)

Dr. Abdul Haseeb Malik

Introduction

- RMI is a Java-implementation of RPC (Remote Procedure Call)
- RMI server
  - creates remote objects
  - makes references to those objects accessible by publishing them using an RMI registry.
  - waits for clients to invoke methods on those objects
- RMI client
  - can locate remote objects from RMI registry and obtain the remote reference to remote objects on a server
  - invokes methods
- Remote communication looks like regular Java method invocations to the programmer and details of remote communication between server and client are handled by RMI


2. Client makes a Naming.lookup call
RMI registry
1. Server registers a remote object, bound to a name
RMI client
3. The registry returns an instance of the remote object's stub
Server that exported a remote object
java.rmi.server.codebase= http://myHost/mydir/
4. Client requests the stub class from the codebase
myHost
5. The HTTP server returns the remote object's stub class
URL location (file, ftp, or http)

Simple Example

- In the following slides a simple example of Java RMI is presented in which:
  - Only a single add method will be provided on the Server
  - The client will locate the object remotely and call this method to add two numbers.

Simple Example: Remote Interface

```
public interface RemMethodInt extends java.rmi.Remote {
        public int add(int a, int b) throws RemoteException;
}
```

- This interface needs to be present at both sides (server and client).
- It tells both sides which methods can be remotely accessible. Here it is only one method which is named *add.*
- The java.rmi.Remote interface is a marker interface which is empty and is only used to let the JVM know that these methods will be used for remote communication.

Simple Example: RMI Server

```
public class RemImpl extends UnicastRemoteObject implements RemMethodInt{
//Constructor should be explicitly called.
//It creates and exports a remote object.
        protected RemImpl() throws RemoteException {super();}
// Implementation of the add method defined in the interface.
        public int add(int a, int b) throws RemoteException {return((a+b));          }

        public static void main(String[] args) {
try {   RemImpl obj = new RemImpl();
        Registry reg = LocateRegistry.createRegistry(5556);
        reg.bind("adder", obj);}
         catch (RemoteException e) {e.printStackTrace();}
         catch (AlreadyBoundException e) {        e.printStackTrace();}
        }
}
```

Simple Example: RMI Client

```
public class clientImp {
	public static void main(String[] args) {
try {
	Registry reg = LocateRegistry.getRegistry("192.168.0.1",5556);
	RemMethodInt obj = (RemMethodInt)reg.lookup("adder");
	System.out.println(obj.add(7, 5));
}
catch (RemoteException | NotBoundException e) {e.printStackTrace();}
	}

}
```

Important Notes for RMI Server in Example

- A class “RemImpl” extends the java.rmi.server.UnicastRemoteObject to provide support for creating and exporting remote objects.
- It also implements the “RemMethodInt” interface and implements the methods defined in the interface.
- At the server side an object of “RemImpl” is created.
- A Registry is created using

```
Registry reg = LocateRegistry.createRegistry(5556);
```

Note: In this case no need to run the registry explicitly

- The object reference will be made available at the following URL address “adder”

```
reg.bind("adder", obj);
```

Important Notes for RMI Client in Example

- We locate the registry which contains references of remote objects on the server system.

Registry reg = LocateRegistry.getRegistry("192.168.0.1",5556);

- Within the registry the client can lookup for the adder object and return the reference of the object.

RemMethodInt obj = (RemMethodInt)reg.lookup("adder");

- Once the client has reference of remote object, it can call methods on it as if it was a local object.

System.out.println(obj.add(7, 5));

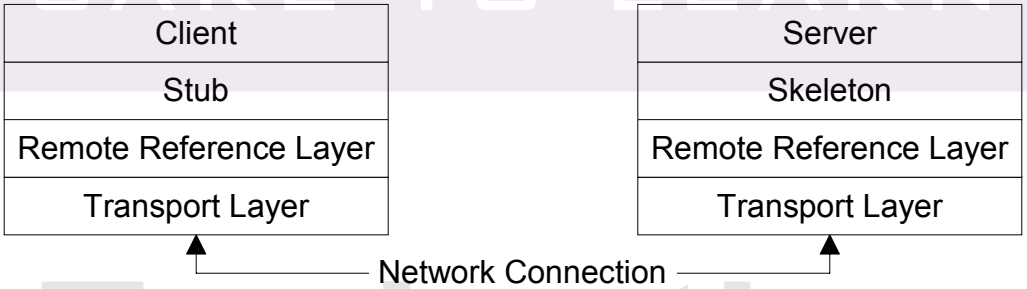
Chapter 13:
Remote Method Invocation (RMI)

Introduction

- The **R**emote **M**ethod **I**nvocation (**RMI**) model represents a **distributed object application.**
  - RMI allows an object inside a JVM (a client) to invoke a method on an object running on a remote JVM (a server) and have the results returned to the client.
    - Therefore, RMI implies a client and a server.
- The server application typically creates an object and makes it accessible remotely.
  - Therefore, the object is referred to as a remote object.
  - The server registers the object that is available to clients.
    - One of the ways this can be accomplished is through a naming facility provided as part of the JDK, which is called the `rmiregistry`.
    - The server uses the registry to bind an arbitrary name to a remote object.
- A client application receives a reference to the object on the server and then invokes methods on it.
  - The client looks up the name in the registry and obtains a reference to an object that is able to interface with the remote object.
    - The reference is referred to as a remote object reference.
  - Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

RMI Architecture

- The interface that the client and server objects use to interact with each other is provided through stubs/skeleton, remote reference, and transport layers.
  - Stubs and skeletons are Java objects that act as proxies to the client and server, respectively.
    - All the network-related code is placed in the stub and skeleton, so that the client and server will not have to deal with the network and sockets in their code.
  - The remote reference layer handles the creation of and management of remote objects.
  - The transport layer is the protocol that sends remote object requests across the network.
- A simple diagram showing the above relationships is shown below.



The Remote Interface

- The server's job is to accept requests from a client, perform some service, and then send the results back to the client.
  - The server must specify an interface that defines the methods available to clients as a service.
    - This **remote interface** defines the client view of the remote object.
- The remote interface is always written to extend the `java.rmi.Remote` interface.
  - `Remote` is a "marker" interface that identifies interfaces whose methods may be invoked from a non-local virtual machine.

CalendarTask.java

```
1.  package examples.rmi;
2.  import java.rmi.Remote;
3.  import java.rmi.RemoteException;
4.  import java.util.Calendar;
5.  public interface CalendarTask extends Remote {
6.      Calendar getDate() throws RemoteException;
7.  }
```

- In the example above, `getDate()` is a remote method of the remote interface `CalendarTask`.
  - All methods defined in the remote interface are required to state that they throw a `RemoteException`.
    - A `RemoteException` represents communication-related exceptions that may occur during the execution of a remote method call.

The Remote Object

- An implementation of the `CalendarTask` interface is shown below.
  - The implementation is referred to as the remote object.
  - The implementation class extends `UnicastRemoteObject` to link into the RMI system.
    - This is not a requirement. A class that does not extend `UnicastRemoteObject` may use its `exportObject()` method to be linked into RMI.
  - When a class extends `UnicastRemoteObject`, it must provide a constructor declaring that it may throw a `RemoteException` object.
    - When this constructor calls super(), it activates code in `UnicastRemoteObject`, which performs the RMI linking and remote object initialization.

CalendarImpl.java

```
 1. package examples.rmi;
 2. import java.rmi.RemoteException;
 3. import java.rmi.server.UnicastRemoteObject;
 4. import java.util.Calendar;
 5.
 6. public class CalendarImpl extends UnicastRemoteObject
 7.         implements CalendarTask {
 8.
 9.     private int counter = 1;
10.
11.     public CalendarImpl() throws RemoteException {}
12.
13.     public Calendar getDate() throws RemoteException{
14.         System.out.print("Method called on server:");
15.         System.out.println("counter = " + counter++);
16.         return Calendar.getInstance();
17.     }
18. }
```

Writing the Server

- The server creates the remote object, registers it under some arbitrary name, then waits for remote requests.
  - The `java.rmi.registry.LocateRegistry` class allows the RMI registry service (provided as part of the JVM) to be started within the code by calling its `createRegistry` method.
    - This could have also been achieved by typing the following at a command prompt: `rmiregistry`.
    - The default port for RMI is `1099`.
  - The `java.rmi.registry.Registry` class provides two methods for binding objects to the registry.
    - `Naming.bind("`*`ArbitraryName`*`", `*`remoteObj`*`);` throws an Exception if an object is already bound under the "*`ArbitrayName`*. "
    - `Naming.rebind ("`*`ArbitraryName`*`", `*`remoteObj`*`);` binds the object under the "*`ArbitraryName`*" if it does not exist or overwrites the object that is bound.
- The example on the following page acts as a server that creates a `CalendarImpl` object and makes it available to clients by binding it under a name of "`TheCalendar`. "

Writing the Server

CalendarServer.java

```
 1. package examples.rmi;
 2. import java.rmi.Naming;
 3. import java.rmi.registry.LocateRegistry;
 4.
 5. public class CalendarServer {
 6.
 7.     public static void main(String args[]) {
 8.         System.out.println("Starting server...");
 9.         // Start RMI registry service and bind
10.         // object to the registry
11.         try {
12.             LocateRegistry.createRegistry(1099);
13.             Naming.rebind("TheCalendar",
14.                     new CalendarImpl());
15.         } catch (Exception e) {
16.             e.printStackTrace();
17.             System.exit(1);
18.         }
19.         System.out.println("Server ready");
20.     }
21. }
```

- If both the client and the server are running Java SE 5 or higher, no additional work is needed on the server side.
  - Simply compile the `CalendarTask`, `CalendarImpl`, and `CalendarServer`, and the server can then be started.
  - The reason for this is the introduction in Java SE 5 of dynamic generation of stub classes.
    - Java SE 5 adds support for the dynamic generation of stub classes at runtime, eliminating the need to use the RMI stub compiler, `rmic`, to pre-generate stub classes for remote objects.
    - Note that `rmic` must still be used to pre-generate stub classes for remote objects that need to support clients running on earlier versions.

The RMI Compiler

- If RMI is being used with a version of Java prior to Java SE 5, a stub must be generated on the server-side and made available to the client.
  - The RMI compiler (`rmic`) is a tool used to create any necessary stubs and/or skeletons to support the remote object.
    - Skelton(s) have been optional since Java SE 1.2.
  - The `rmic` compiler is passed to the compiled version of the remote object and generates a stub class from it as shown below.

```
rmic -keep -d %CLASSES% examples.rmi.CalendarImpl
```

    - The "`-keep`" is optional and is being used so that, in addition to the generated `.class` files, the `.java` files will be retained so that they can be viewed if desired.
    - The result of running the `rmic` command above is a file named `CalendarImpl_Stub.class`.
    - Since the "`-keep`" option was used, there is also a file named `CalendarImpl_Stub.java`.
  - The `CalendarImpl_Stub` class generated is a client-side component and as such, must exist on the client's classpath in order for client code to successfully communicate with the server.
    - If the stub class is not available locally to the client, it must be loaded dynamically over the network.
    - Keep in mind that this is only necessary if a version of Java prior to Java SE 5 is being used.

Writing the Client

- An RMI client is a program that accesses the services provided by a remote object.
  - The `java.rmi.registry.LocateRegistry` class allows the RMI registry service to be located by a client by its `getRegistry` method.
    - The `java.rmi.registry.Registry` class provides a `lookup` method that takes the "`ArbitraryName`" the remote object was bound to by the server.
- Once the client obtains a reference to a remote object, it invokes methods as if the object were local.

CalendarClient.java

```
 1. package examples.rmi;
 2.
 3. import java.rmi.registry.*;
 4. import java.util.Calendar;
 5.
 6. public class CalendarClient {
 7.
 8.     public static void main(String args[]) {
 9.         Calendar c = null;
10.         CalendarTask remoteObj;
11.         String host = "localhost";
12.         if(args.length == 1)
13.             host = args[0];
14.         try {
15.             Registry r =
16.               LocateRegistry.getRegistry(host, 1099);
17.             Object o = r.lookup("TheCalendar");
18.             remoteObj = (CalendarTask) o;
19.             c = remoteObj.getDate();
20.         } catch (Exception e) {
21.             e.printStackTrace();
22.         }
23.         System.out.printf("%tc", c);
24.     }
25. }
```

Remote Method Arguments and Return Values

- The arguments to a remote method must be `Serializable`.
  - They must be primitive types or objects that implement the `Serializable` interface.
  - The same restriction applies to return values.
- The RMI stub/skeleton layer decides how to send arguments and return values over the network.
  - If the object is `Serializable` but not `Remote`:
    - the object is serialized and streamed in byte format; and
    - the receiver de-serializes the bytes into a copy of the original object.
  - If the object is a `Remote` object:
    - a remote reference for the object is marshaled and sent to the remote process; and
    - this reference is received and converted into a stub for the original object.
  - If the argument or return value is not serializable, a `java.rmi.MarshalException` is thrown.
- The key difference between remote and non-remote objects is that `Remote` objects are sent by reference, while non-remote objects (and primitive types) are sent by copy.

Dynamic Loading of Stub Classes

- If the client stub class is not available in the local `CLASSPATH`, it must be loaded dynamically over the network.
  - This is a typical scenario when the client and server are not running on the same machine.
  - We will illustrate downloading of stub classes via a web server.
- When the RMI run-time system marshals a remote object stub, it encodes a URL in the byte stream to tell the process on the other end of the stream where to look for the class file for the marshaled object.
  - This URL is obtained from a system property called `java.rmi.server.codebase`.
    - We will set this property on the command line to point to a directory within the web server's document base.
  - Note that in order for a Java runtime system to be able to load classes remotely, it has to have a security manager installed that will allow the remote load.
    - There is one provided by the `java.rmi.RMISecurityManager` class.
  - The final issue is that the default Java security policy does not allow all the networking operations required to load a class from a remote host.
    - An RMI client that needs to load classes remotely must have a policy file granting the necessary permissions.
    - The name of the policy file can be specified on the command line by setting the `java.security.policy` property.

Remote RMI Client Example

- The RMI application shown below illustrates dynamic loading of stub classes.
  - It also shows an example of a `Remote` object used as a method argument.
  - Following the source code are detailed instructions on how to run the Client and Server.

- We begin with the remote interface.

Account.java

```
 1. package examples.rmi;
 2.
 3. import java.rmi.*;
 4.
 5. public interface Account extends Remote {
 6.     public String getName() throws RemoteException;
 7.
 8.     public double getBalance()
 9.       throws RemoteException;
10.
11.     public void withdraw(double amt)
12.       throws RemoteException;
13.
14.     public void deposit(double amt)
15.       throws RemoteException;
16.
17.     public void transfer(double amt, Account src)
18.       throws RemoteException;
19. }
```

- The implementation of the remote interface is shown on the next page.

Remote RMI Client Example

AccountImpl.java

```
 1. package examples.rmi;
 2.
 3. import java.rmi.server.*;
 4. import java.rmi.*;
 5.
 6. public class AccountImpl extends UnicastRemoteObject
 7.   implements Account {
 8.     private double balance = 0.0;
 9.     private String name = "";
10.
11.     public AccountImpl(String aName)
12.       throws RemoteException {
13.         name = aName;
14.     }
15.
16.     public String getName() throws RemoteException {
17.         return name;
18.     }
19.
20.     public double getBalance()
21.       throws RemoteException {
22.         return balance;
23.     }
24.
25.     public void withdraw(double amt)
26.       throws RemoteException {
27.         if (amt > balance)
28.             throw new RemoteException();
29.         balance -= amt;
30.     }
31.
32.     public void deposit(double amt)
33.       throws RemoteException {
34.         balance += amt;
35.     }
36.
37.     public void transfer(double amt, Account src)
38.       throws RemoteException {
39.         src.withdraw(amt);
40.         this.deposit(amt);
41.     }
42. }
```

Remote RMI Client Example

- The Server is shown below with the following features.
  - The compiling of the stub class for the client is done using `Runtime.exec()`.
    - The `exec` method allows the JVM to run an external process (in this case the rmi compiler - `rmic`).
  - The server creates and starts the registry service.

AccountServer.java

```
 1. package examples.rmi;
 2.
 3. import java.io.IOException;
 4. import java.net.InetAddress;
 5. import java.rmi.registry.*;
 6.
 7. public class AccountServer {
 8.     private static String buildCommandLine(){
 9.         String jcp = "java.class.path";
10.         StringBuffer sb = new StringBuffer();
11.         sb.append('"');
12.         sb.append(System.getProperty(jcp));
13.         sb.append('"');
14.         String classpath = sb.toString();
15.         sb.setLength(0);
16.         sb.append("rmic -d ").append(classpath);
17.         sb.append(" -classpath ").append(classpath);
18.         sb.append(" examples.rmi.AccountImpl");
19.         System.out.println(sb.toString());
20.         return sb.toString();
21.     }
22.     public static void main(String args[]) {
23.         // execute rmic as an external process
24.         Process p = null;
25.         try {
26.             String command = buildCommandLine();
27.             p = Runtime.getRuntime().exec(command);
28.             p.waitFor(); // wait for completion
29.         } catch (Exception e1) {
30.             e1.printStackTrace();
31.         }
```

Remote RMI Client Example

AccountServer.java - continued

```
32.
33.         try {
34.             String key = "java.rmi.server.codebase";
35.             InetAddress server =
36.                 InetAddress.getLocalHost();
37.             String address = server.getHostAddress();
38.             String value =
39.                 "http://" + address + ":8080/";
40.             System.setProperty(key, value);
41.             //Start the StubServer
42.             Thread t = new StubServer();
43.             t.start();
44.             // Create registry service
45.             Registry reg =
46.                 LocateRegistry.createRegistry(1099);
47.             // Create some Accounts
48.             AccountImpl acct1 =
49.               new AccountImpl("Alan");
50.             AccountImpl acct2 =
51.               new AccountImpl("Dave");
52.
53.             // Register with the naming registry.
54.             reg.rebind("Alan", acct1);
55.             reg.rebind("Dave", acct2);
56.
57.             System.out.println("Accts registered");
58.         } catch (Exception e) {
59.             e.printStackTrace();
60.         }
61.     }
62. }
```

Remote RMI Client Example

- Although a web server such as Tomcat, WebLogic, or WebSphere could be used to host the stub class necessary for dynamic loading of the stub class, the file below is a simple web server based on the code from the Networking chapter of this course.

StubServer.java

```
 1. package examples.rmi;
 2.
 3. import java.io.*;
 4. import java.net.*;
 5.
 6. public class StubServer extends Thread {
 7.
 8.     static byte[] hdrNotFound =
 9.         "HTTP/1.0 404 Not Found\n\n".getBytes();
10.     static byte[] notFound =
11.         "<html>Resource Not Found</html>".getBytes();
12.     static byte[] hdrOK =
13.         "HTTP/1.0 200 OK\n\n".getBytes();
14.     static byte[] testResponse =
15.         "<html>Server operational</html>".getBytes();
16.
17.     public void run() {
18.         ServerSocket theServer = null;
19.         Socket clientSocket;
20.         // Attempt to start the server
21.         try {
22.             theServer = new ServerSocket(8080);
23.             while (true) {
24.                 clientSocket = theServer.accept();
25.                 handleClient(clientSocket);
26.             }
27.         } catch (IOException ioe) {
28.             ioe.printStackTrace();
29.             System.exit(1);
30.         }
31.     }
32.
```

▸ *Continued on following page*

Remote RMI Client Example

StubServer.java - *continued*

```
33.     private void handleClient(Socket cSocket) {
34.         OutputStream toClient = null;
35.         BufferedReader fromClient = null;
36.         try {
37.             // Get Input and Output
38.             fromClient = new BufferedReader(
39.                     new InputStreamReader(cSocket
40.                      .getInputStream()));
41.             toClient = cSocket.getOutputStream();
42.             // read from Client
43.             String theLine = fromClient.readLine();
44.             String request = theLine.split(" ")[1];
45.             System.out.println("StubServer Request:"
46.                                + theLine);
47.             if (request.equals("/")) {
48.                 toClient.write(hdrOK);
49.                 toClient.write(testResponse);
50.             } else {
51.                 processStub(request, toClient);
52.             }
53.
54.             fromClient.close();
55.             toClient.close();
56.             cSocket.close();
57.         } catch (IOException ioe) {
58.             String msg = "Connection lost";
59.             System.out.println(msg);
60.         }
61.     }
```

▸ *Continued on following page*

Remote RMI Client Example

StubServer.java - *continued*

```
62.     private void processStub(String req,
63.             OutputStream toClient){
64.         InputStream is =
65.             this.getClass().getResourceAsStream(req);
66.
67.         byte[] bufferedStub = null;
68.         try {
69.             if (is != null) {
70.                 int size = is.available();
71.                 bufferedStub = new byte[size];
72.                 is.read(bufferedStub);
73.                 is.close();
74.                 toClient.write(hdrOK);
75.                 toClient.write(bufferedStub);
76.             } else {
77.                 toClient.write(hdrNotFound);
78.                 toClient.write(notFound);
79.             }
80.         } catch (IOException e) {
81.             e.printStackTrace();
82.         }
83.     }
84. }
```

- Finally, the client code is shown below.

AccountClient.java

```
 1. package examples.rmi;
 2.
 3. import java.net.URL;
 4. import java.rmi.*;
 5. import java.rmi.registry.*;
 6.
 7. public class AccountClient {
 8.     public static void main(String args[]) {
 9.         String host = "localhost";
10.         if (args.length > 0) { host = args[0]; }
11.         try {
12.             String key = "java.security.policy";
13.             Class c = AccountClient.class;
14.             URL u =c.getResource("policy.client");
```

Remote RMI Client Example

AccountClient.java - *continued*

```
15.             String value = u.getPath();
16.             System.setProperty(key, value);
17.             System.setSecurityManager(
18.                     new RMISecurityManager());
19.             Registry r =
20.              LocateRegistry.getRegistry(host, 1099);
21.             // Lookup Account objects
22.             Account act1 =
23.                 (Account) r.lookup("Alan");
24.             Account act2 =
25.                 (Account) r.lookup("Dave");
26.
27.             showBalance(act1);
28.             showBalance(act2);
29.
30.             // Make some deposits
31.             act1.deposit(200);
32.             act2.deposit(100);
33.
34.             // Show results
35.             System.out.println("Deposit 200 & 100");
36.             showBalance(act1);
37.             showBalance(act2);
38.
39.             // Do a transfer
40.             act2.transfer(10, act1);
41.
42.             // Show results
43.             System.out.println("Transfer 10");
44.             showBalance(act1);
45.             showBalance(act2);
46.         } catch (Exception e){ e.printStackTrace(); }
47.     }
48.
49.     public static void showBalance(Account acct)
50.       throws RemoteException {
51.         System.out.println("Balance for " +
52.                 acct.getName() + " is " +
53.                 acct.getBalance());
54.     }
55. }
```

Running the Remote RMI Client Example

- Running the `AccountServer` will accomplish the following.
  - It generates the `AccountImpl_Stub` needed by the client so that it can be dynamically loaded by the client.
  - It starts the `StubServer` so that the `AccountImpl_Stub` is available via the `java.rmi.server.codebase` property.
  - It starts the rmi registry to bind the remote `Account` objects.
- Running the `AccountClient` will accomplish the following.
  - It installs the `RMISecurityManager`.
  - It utilizes the policy file named `policy.client` to allow the JVM to load a class from a remote URL.
  - If the Stub class is not available on the client's classpath when the lookup is performed, the client will automatically retrieve the necessary stub from the servers codebase.

Exercises

1. Build and test an implementation for a `MathServices` interface with remote methods as shown below.

```
public double sqroot (double value);
public double square(double value);
```

- Begin with `MathServices.java` in the `starters` directory.

This Page Intentionally Left Blank