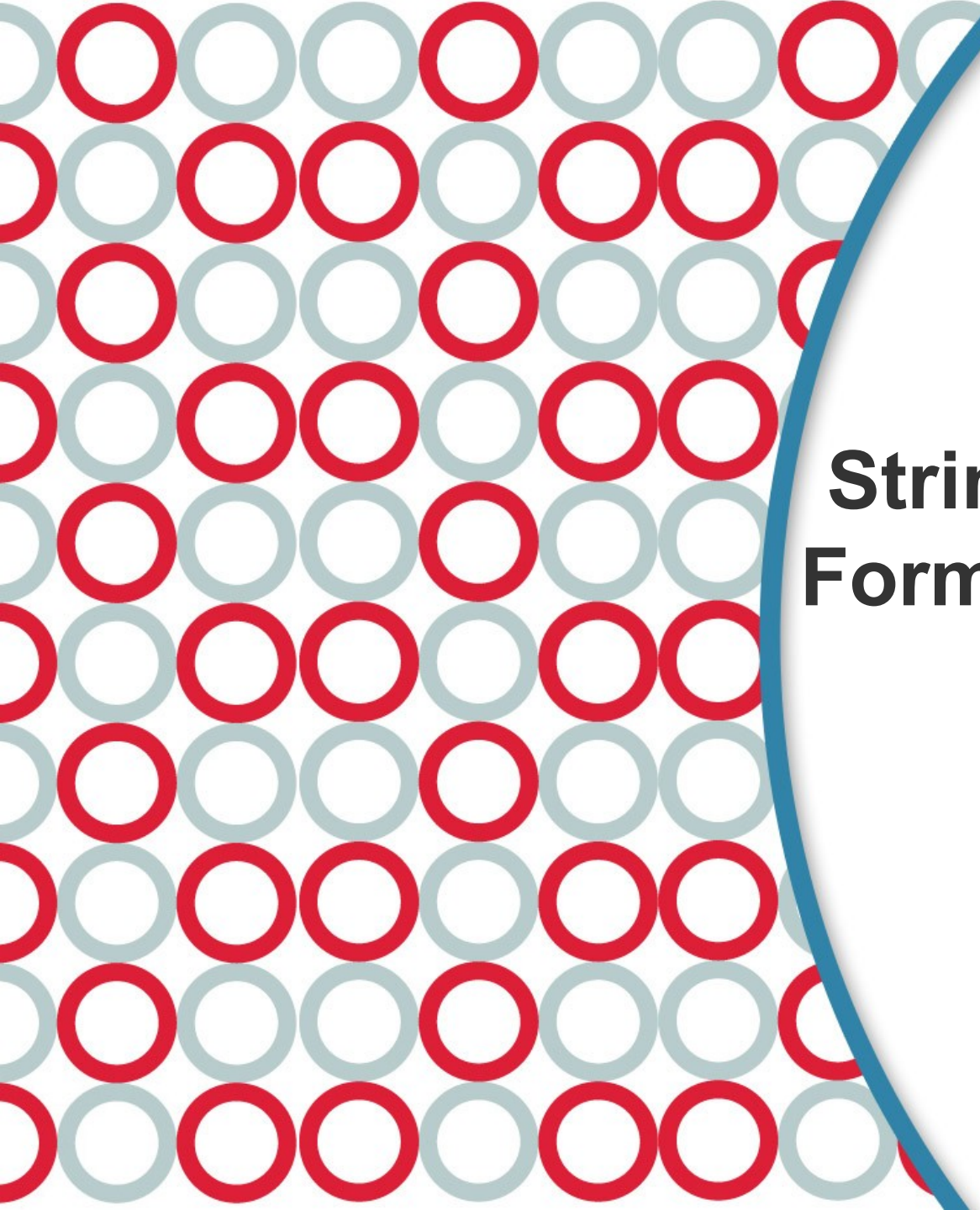


May 2019, IPT Course
Java Web Debelopment

JWD: OOP, String Processing, Formatting, RegEx, Resources

Trayan Iliev
tiliev@iproduct.org
http://iproduct.org

About me

Trayan Iliev

– CEO of IPT – Intellectual Products & Technologies

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6/7, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer

– Voxxed Days, jPrime, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast (http://robolearn.org)

Where to Find the Code?

Java Web Development projects and examples are available @ GitHub:

https://github.com/iproduct/course-java-web-development

Agenda for This Session

- OOP principles – Encapsulation, Inheritance and Polymorphism, Overriding / Overloading
- String Processing,
- Data Formatting, Resource Bundles, Regular Expressions
- java.util & java.math
- StringTokenizer, Date/Calendar,
- Locale, Random, Optional, Observable, Observable interface, BigDecimal

Basic Concepts in OOP and OOAD

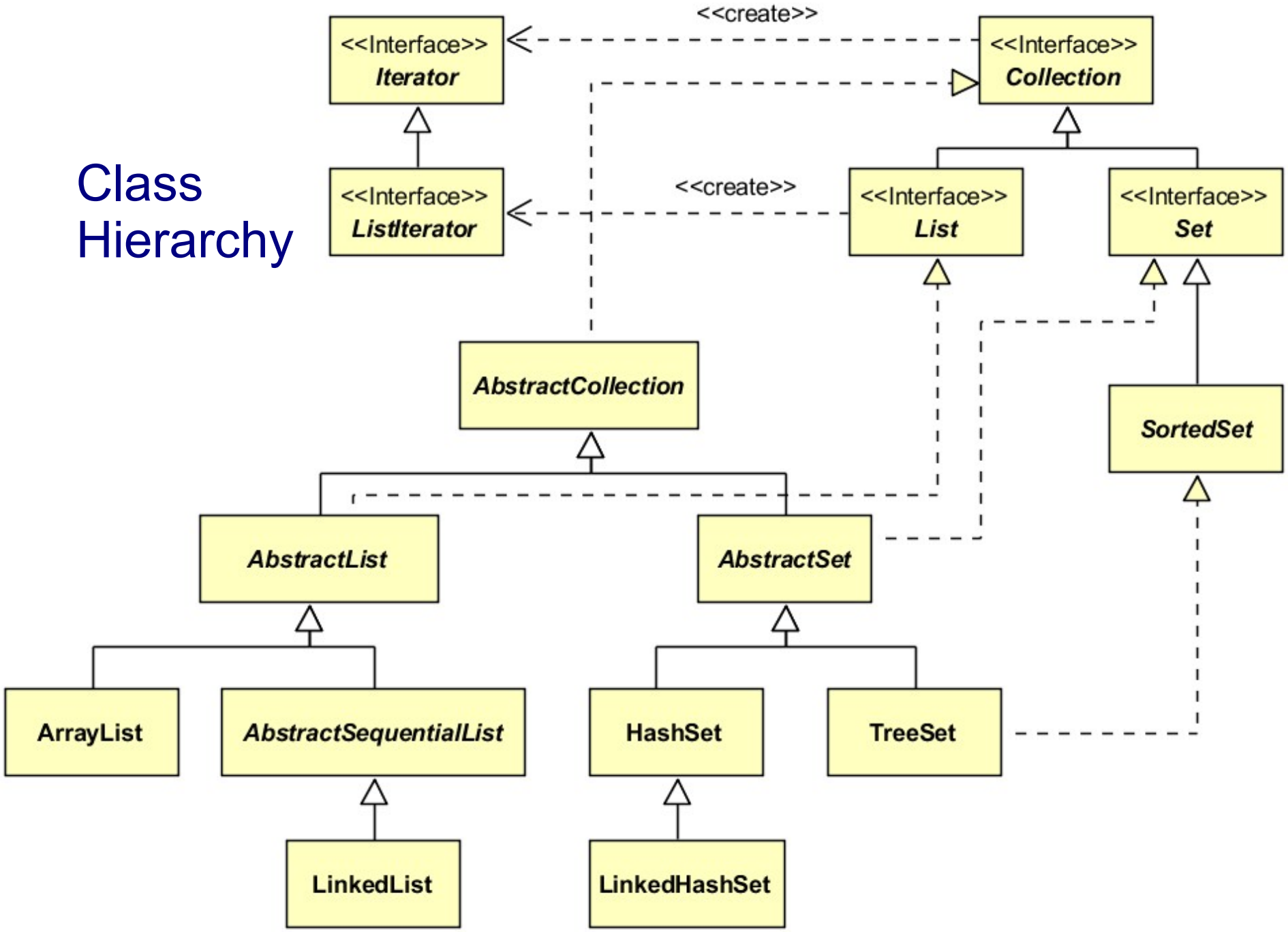
- interface and implementation – we divide what remains constant (contractual interface) from what we would like to keep our freedom to change (hidden realization of this interface)
- interface = public
- implementation = private
- This separation allows the system to evolve while maintaining backward compatibility to already implemented solutions, enables parallel development of multiple teams
- programming based on contractual interfaces

Object-Oriented Approach to Programming

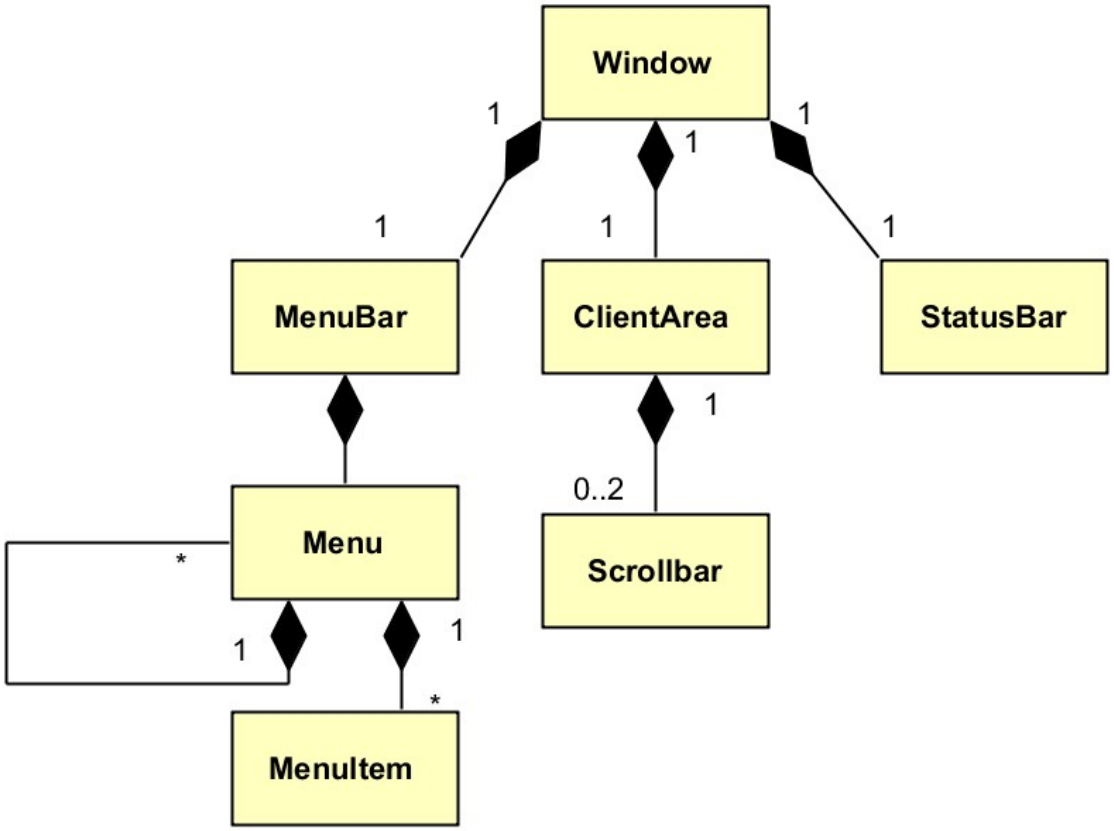
Key elements of the object model [Booch]:

- class, object, interface and implementation
- abstraction – basic distinguishing characteristics of an object
- capsulation – separating the elements of abstraction that make up its structure and behavior - interface and implementation
- modularity – decomposing the system into a plurality of components and loosely connected modules - principle: maximum coherence and the minimum connectivity
- hierarchy – class and object hierarchies

Class Hierarchy



Object Hierarchy



Object-Oriented Approach to Programming

Additional elements of the object model [Booch]:

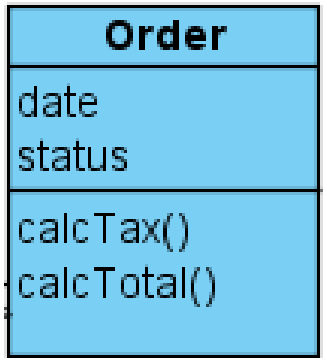
- typing – requirement for the class of an object such that objects of different types can not be replaced (or can in a strictly limited way)
  - static and dynamic binding
  - polymorphism
- concurrency – abstraction and synchronization of processes
- length of life – object-oriented databases

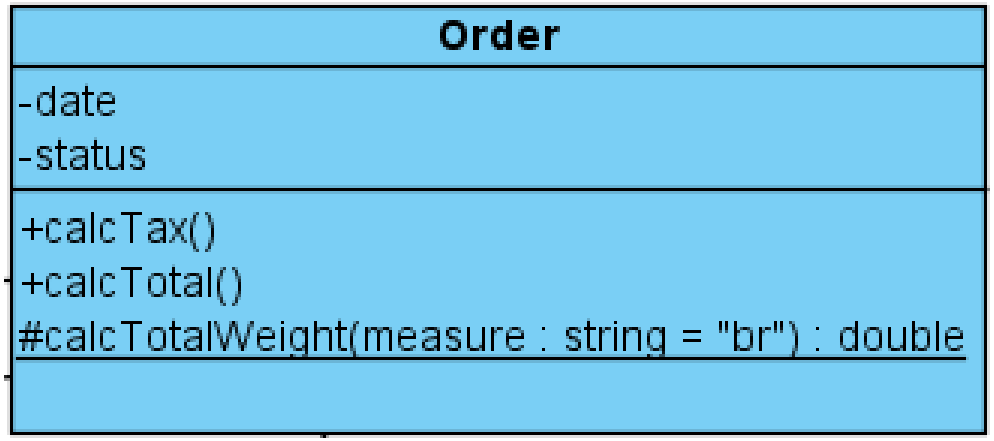
Classes

Class – describes a set of objects that share the same specifications of the characteristics (attributes and methods), constraints and semantics

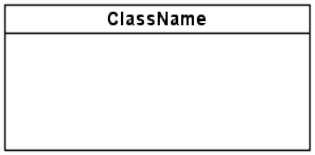
- attributes – instances of properties in UML, they can provide end of association, object *structure*
- operations - behavioral characteristics of a classifier, specifying name, type, parameters and constraints for invoking definitely associated with the operation behavior

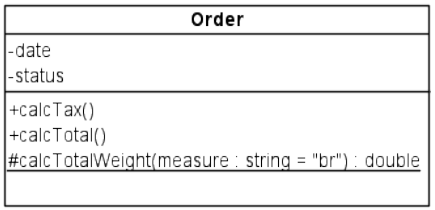
Classes - Graphical Notation in UML

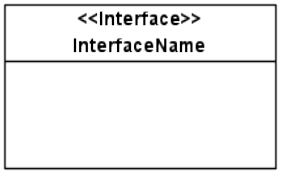


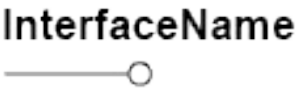


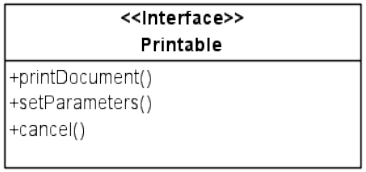
Elements of Class Diagrams





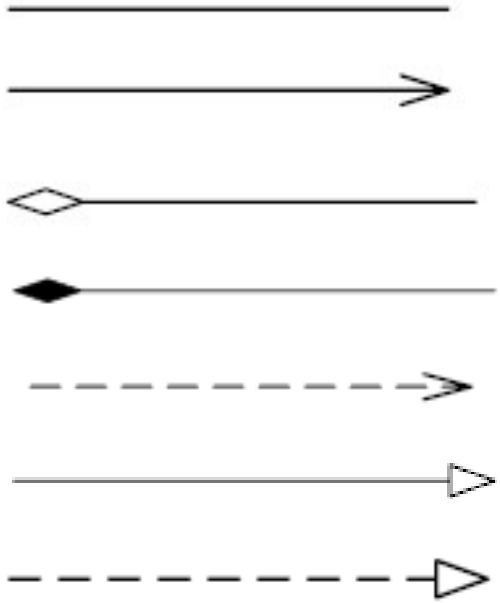




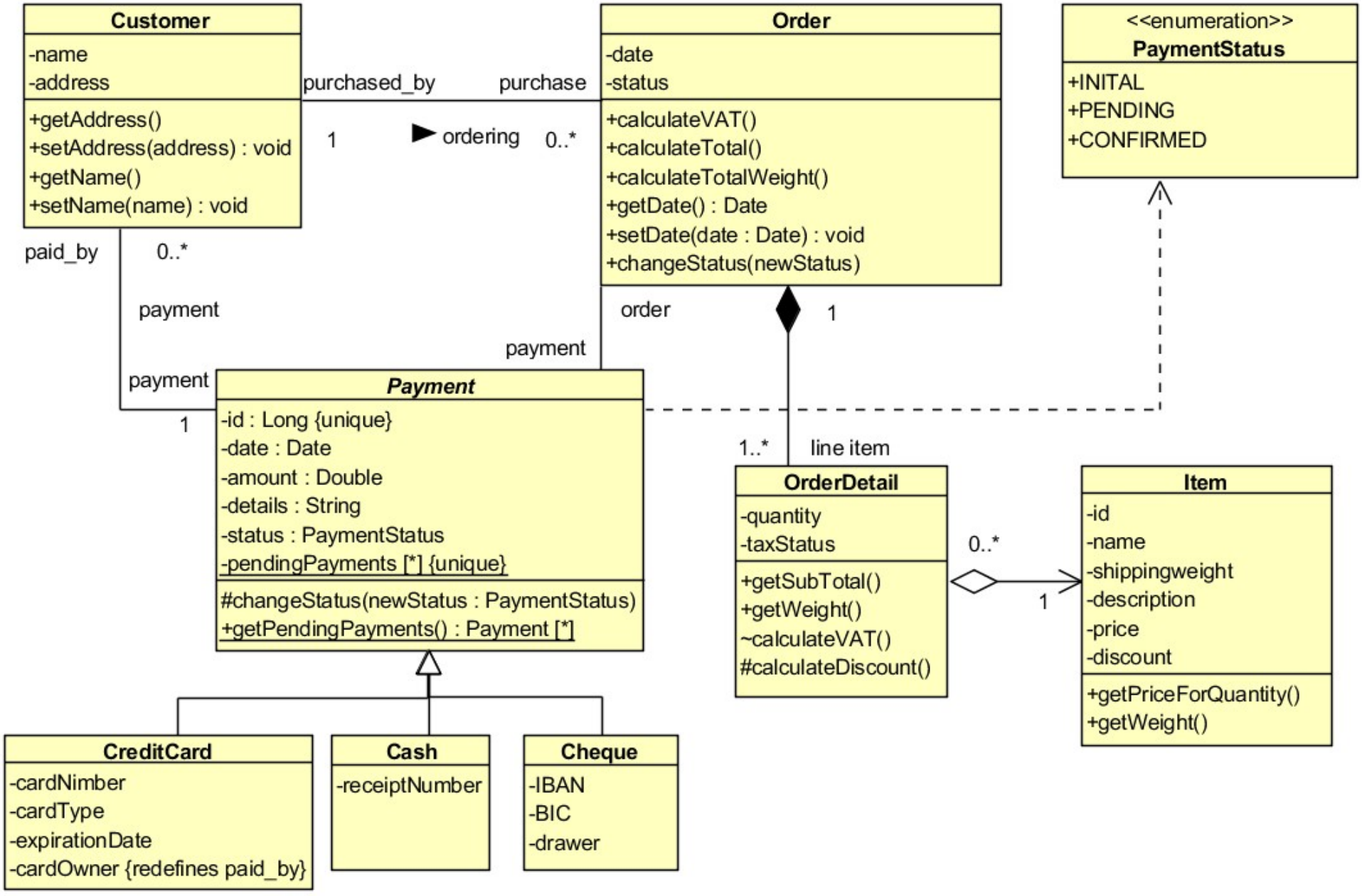


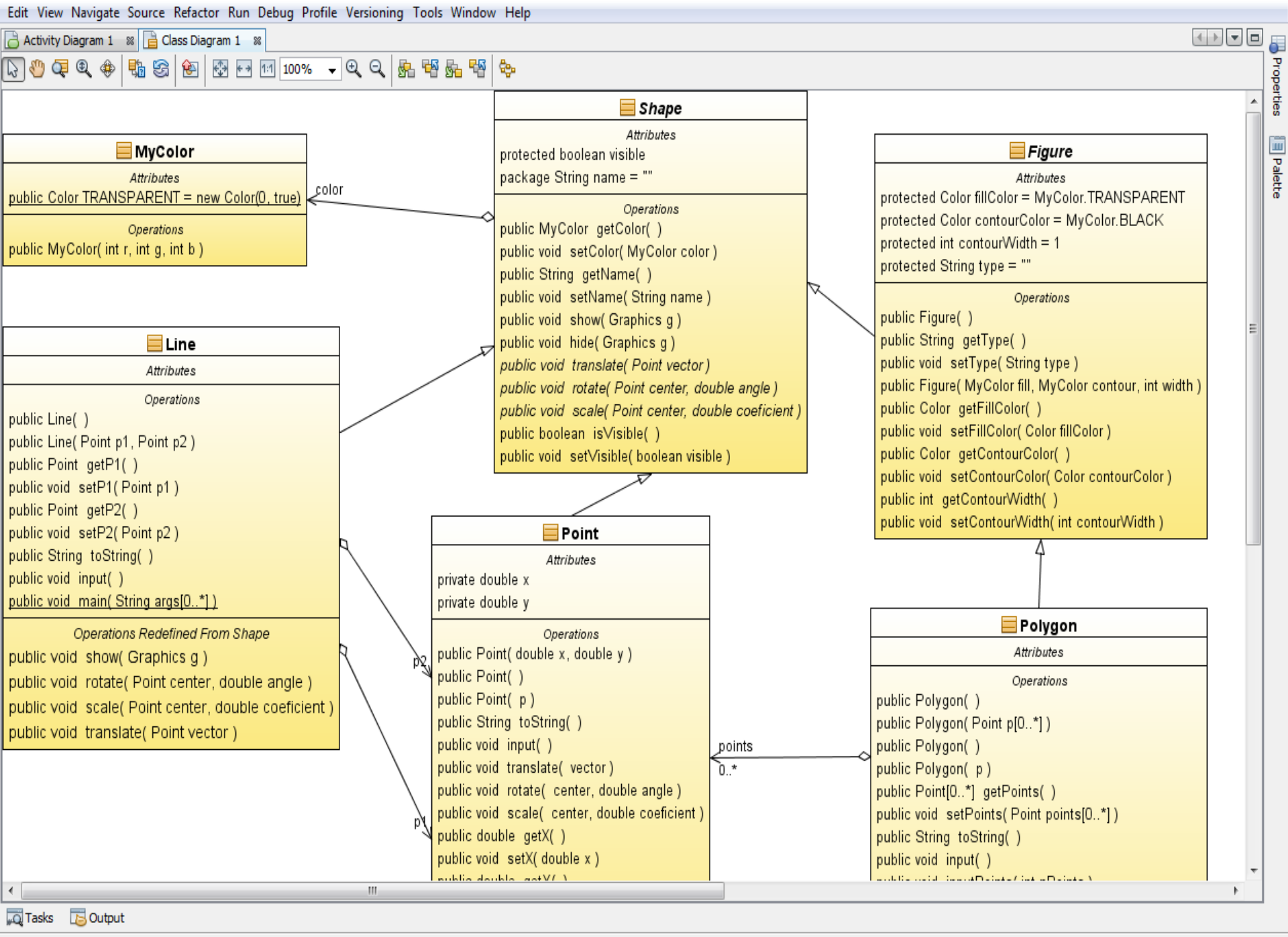
Types of connections:

- association
- aggregation
- composition
- dependence
- generalization
- realization



Class Diagram - 1

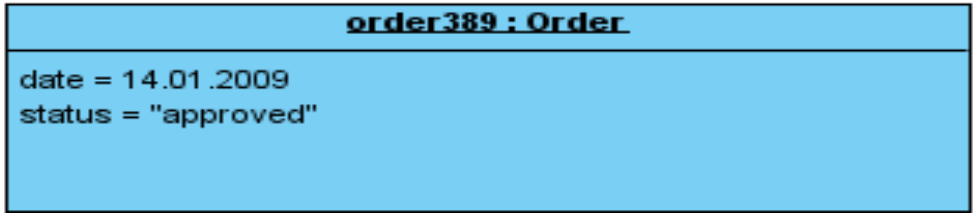



Edit View Navigate Source Refactor Run Debug Profile Versioning Tools Window Help
Activity Diagram 1
Class Diagram 1
100%
Properties
Palette
MyColor
Attributes
public Color TRANSPARENT = new Color(0, true)
Operations
public MyColor(int r, int g, int b)
color
Shape
Attributes
protected boolean visible
package String name = ""
Operations
public MyColor getColor( )
public void setColor(MyColor color)
public String getName( )
public void setName(String name)
public void show(Graphics g)
public void hide(Graphics g)
public void translate(Point vector)
public void rotate(Point center, double angle)
public void scale(Point center, double coeficient)
public boolean isVisible( )
public void setVisible(boolean visible)
Figure
Attributes
protected Color fillColor = MyColor.TRANSPARENT
protected Color contourColor = MyColor.BLACK
protected int contourWidth = 1
protected String type = ""
Operations
public Figure( )
public String getType( )
public void setType(String type)
public Figure(MyColor fill, MyColor contour, int width)
public Color getFillColor( )
public void setFillColor(Color fillColor)
public Color getContourColor( )
public void setContourColor(Color contourColor)
public int getContourWidth( )
public void setContourWidth(int contourWidth)
Line
Attributes
Operations
public Line( )
public Line(Point p1, Point p2)
public Point getP1( )
public void setP1(Point p1)
public Point getP2( )
public void setP2(Point p2)
public String toString( )
public void input( )
public void main(String args[0..*])
Operations Redefined From Shape
public void show(Graphics g)
public void rotate(Point center, double angle)
public void scale(Point center, double coeficient)
public void translate(Point vector)
p2
p1
Point
Attributes
private double x
private double y
Operations
public Point(double x, double y)
public Point( )
public Point(p)
public String toString( )
public void input( )
public void translate(vector)
public void rotate(center, double angle)
public void scale(center, double coeficient)
public double getX( )
public void setX(double x)
points
0..*
Polygon
Attributes
Operations
public Polygon( )
public Polygon(Point p[0..*])
public Polygon( )
public Polygon(p)
public Point[0..*] getPoints( )
public void setPoints(Point points[0..*])
public String toString( )
public void input( )
Tasks
Output

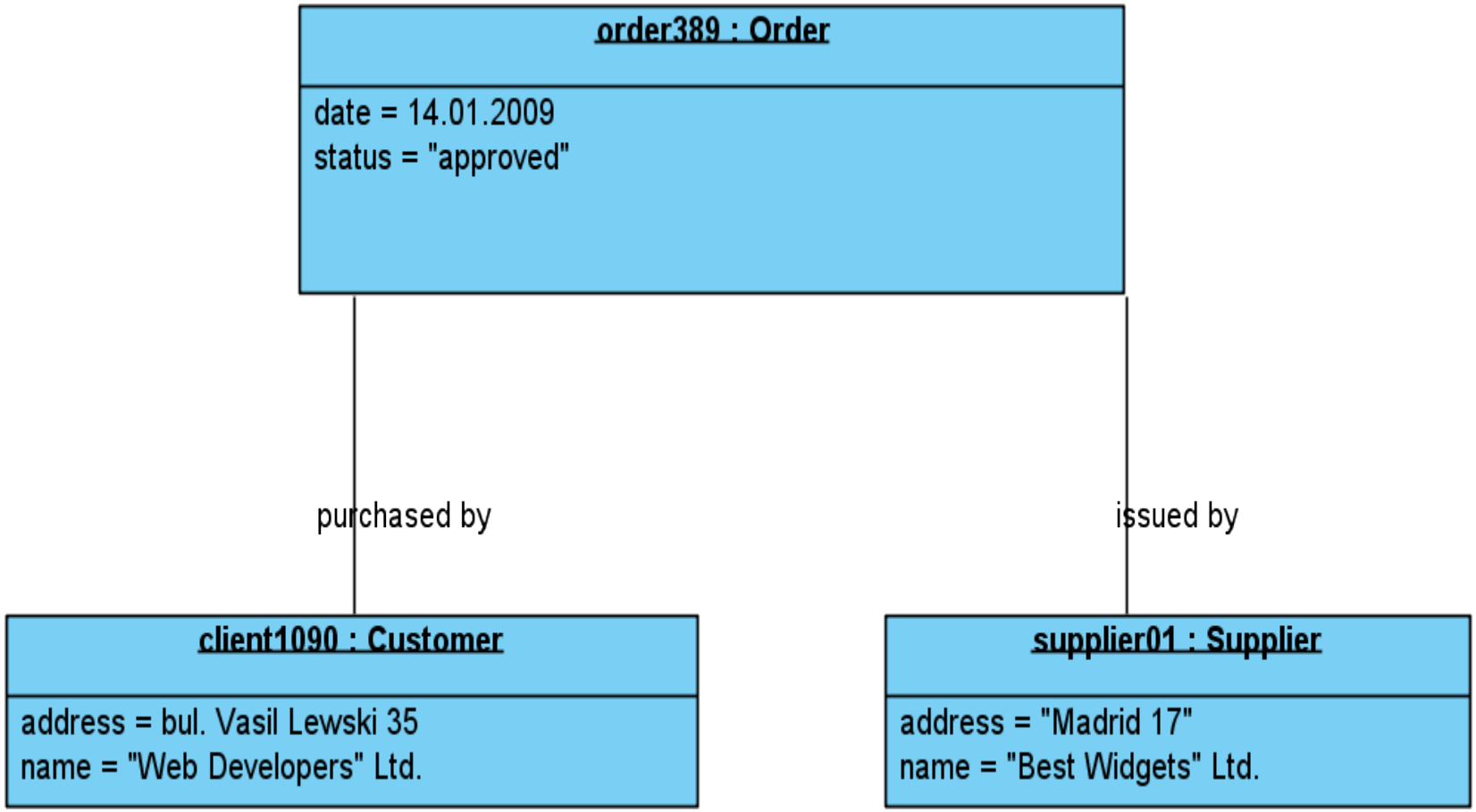
Objects

Instance specification = Object – represents an instance of the modeled system, for example class -> object association -> link, property -> attribute, etc.

- can provide illustration or example of object
- describes the object in a particular moment of time
- may be uncomplete
- Example:



Object Diagram



Analysis Classes Stereotypes

Analysis classes are used in the mapping and analysis of system architecture - they present rather different roles and responsibilities, than specific classes to be realized, and are independent of implementation technology:

- <<controll>> - business logic
- <<entity>> - data
- <<boundary>> - user or system interface

Object Constructors in Java

- Initialization of objects with constructors
- Overloading of constructors and other methods
- Default constructors
- Reference to the current object – **this**

Objects Initialization. Array initialization

- Initialization in declaration
- Initialization in constructor
- „Lazy“ initialization
- Initialization of static class members
- One-dimensional and multi-dimensional arrays
- Array initialization

Strings

- **String** class provides **immutable** objects – i.e. any operation on the string creates a new object in hip
- **StringBulider** – it provides an efficient way from the side of resources to modify the strings, as realize Reusable Design Pattern: **Builder** – for incremental string building (basically with methods append and insert)
- Basic operations in the class **String**. Formatted output - method **format()** and class **Formatter**. Specifiers:

%[argument_index$][flags][width]
[.precision]conversion

Conversion in Type Formatting

- d – decimal, integral types
- c – character (unicode)
- b - boolean
- s - String
- f – float, double (with decimal point)
- e - float, double (scientific notation)
- x – hexadecimal value of integral types
- h – hexadecimal hash code

Regular Expressions - I

- Symbolic classes:
  - **.** Any character (may or may not match line terminators)
  - **\d** A digit: [0-9]
  - **\D** A non-digit: [^0-9]
  - **\s** A whitespace character: [\t\n\x0B\f\r]
  - **\S** A non-whitespace character: [^\s]
  - **\w** A word character: [a-zA-Z_0-9]
  - **\W** A non-word character: [^\w]

Regular Expressions - II

- Qualifiers:
  - **X?** X, once or not at all
  - **X*** X, zero or more times
  - **X+** X, one or more times
  - **X{n}** X, exactly n times
  - **X{n,}** X, at least n times
  - **X{n,m}** X, at least n but not more than m times
- **Greedy, Reluctant (?) & Possessive (+)** qualifiers
- **Capturing Group - (X)**

Regular Expressions - III

- Class **Pattern** – basic methods:
  - **public static Pattern compile(String regex)**
  - **public Matcher matcher(CharSequence input)**
  - **public static boolean matches(String regex, CharSequence input)**
  - **public String[] split(CharSequence input, int limit)**
- Class **Matcher** – basic methods:
  - **public boolean matches()**
  - **public boolean lookingAt()**
  - **public boolean find(int start)**
  - **public int groupCount()** и **public String group(int group)**

Exception Handling in Java

- Obligatory exception handling in Java → secure and reliable code
- Separation of concerns: business logic from exception handling code
- Class **Throwable** → classes Error и Exception
- Generating Exceptions – keyword **throw**
- Exception handling:
  - **try – catch – finally** block
  - Delegating the handling to the caller method - **throws**

Novelties in Java 8+: Date-Time API (JSR 310)

- Allows flexible processing (incl. time based calculations) with dates and periodes
- Package: **java.time**
- Supported standards: **ISO-8601, Unicode Common Locale Data Repository (CLDR), Time-Zone Database (TZDB)**
- Example:

```
LocalDate today = LocalDate.now();
LocalDate reportDay =
today.with(TemporalAdjusters.lastDayOfMonth());
LocalDate paymentDay = reportDay.plusDays(5);
```

Advantages of Date-Time API

- **Clear** - methods in the API are well defined and their behavior is clear and expected
- **Fluent** - provides a fluent interface, making the code easy to read, most methods do not allow null values => can be chained together:

```
LocalDate today = LocalDate.now();
LocalDate reportDay = today
        .with(TemporalAdjusters.lastDayOfMonth())
        .minusDays(2);
```

Advantages of Date-Time API

- **Immutable** - after the object is created, it cannot be modified. To alter the value of an immutable object, a new object must be constructed as a modified copy of the original => thread-safe. This affects the API in that most of the methods used to create date or time objects are prefixed with of, from, or with, rather than constructors, and there are no set methods. For example:

```
LocalDate dateOfBirth =
      LocalDate.of(2012, Month.MAY, 14);
LocalDate firstBirthday = dateOfBirth.plusYears(1);
```

- **Extensible** - wherever possible. For example, you can define your own time adjusters and queries, or build your own calendar system.

Date and Time API: Main Classes (1)

- **Clock** – allows access to the current moment, date and time for a time zone
- **Instant** – momentary point on the time axis
- **LocalDate** –local date without time and zone: 2014-12-20
- **LocalTime** – local time without date and zone: 14:25:15
- **LocalDateTime** – local date and time without zone: 2014-12-20T14:25:15
- **MonthDay** – day of month –12-20 => December 20
- **Duration** – time period – e.g. 2 minuts and 52 seconds
- **Period** – time period in days – e.g. 3 years 2 months and 4 days

Date and Time API: Main Classes (2)

- **OffsetDateTime** – date and time + time zone: 2014-12-20T09:15:00+02:00
- **OffsetTime** – time + time zone: 09:15:00+02:00.
- **Year** – year - e.g. 2014
- **YearMonth** – month in year – напр. 2014-12
- **ZonedDateTime** – similar to OffsetDateTime + time-zone ID: 2014-12-20T10:15:30+01:00 Europe/Sofia
- **ZoneId** – time-zone ID – e.g. Europe/Rome
- **ZoneOffset** – time offset from Greenwich/UTC – e.g. +02:00

Date and Time API – Example

```
LocalDate today = LocalDate.now();
LocalDate dateOfBirth = LocalDate.of(1982, Month.MAY, 14);
LocalDateTime now = LocalDateTime.now();
LocalTime timeOfBirth = LocalTime.of(14, 50);
LocalDateTime dateTimeOfBirth = LocalDateTime.of(dateOfBirth,
timeOfBirth);
Period howOld = Period.between(dateOfBirth, today);
Duration age = Duration.between(dateTimeOfBirth, now);
long daysOld = ChronoUnit.DAYS.between(dateOfBirth, today);
System.out.println("Your age are: " + howOld.getYears() + " years, "
    + howOld.getMonths() + " months, and " + howOld.getDays()
    + " days. (" + age.toDays() + " /" + daysOld + "/ days total)");
```

Property files, ResourceBundles, I18N and L10N

- Property files, XML properties: https://docs.oracle.com/javase/tutorial/essential/environment/properties.html

- MyLabels.properties:

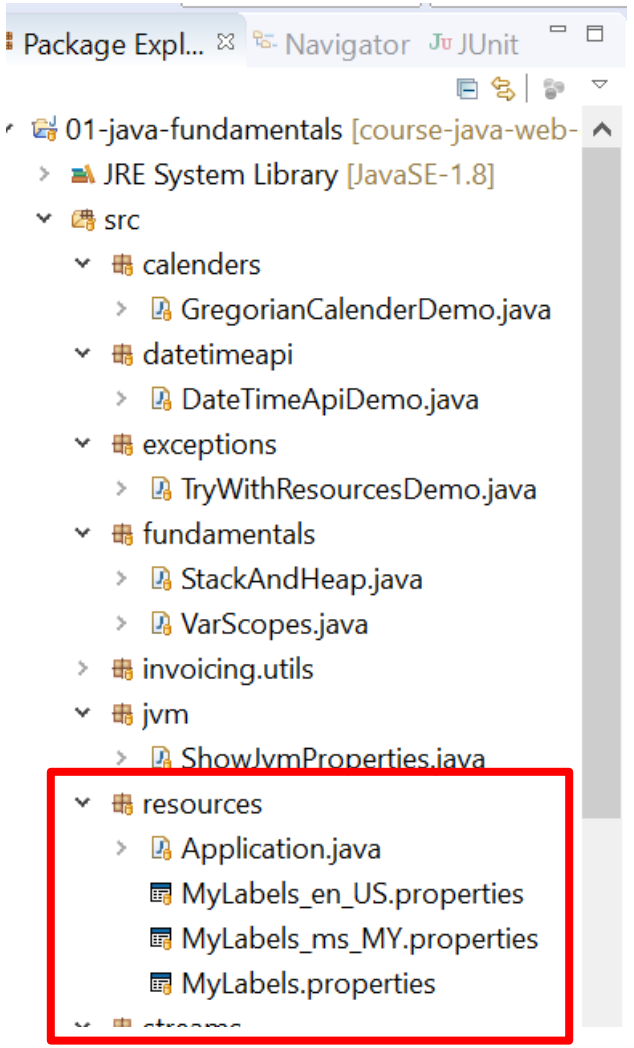
```
how_are_you = How are you?
```

- MyLabels_en_US.properties:

```
how_are_you = How are you?
```

- MyLabels_ms_MY.properties:

```
how_are_you = apa khabar
```



Property files, ResourceBundles, I18N and L10N

- Working with property files, XML properties, classes: https://www.baeldung.com/java-properties
- Example:

```
String rootPath =
Thread.currentThread().getContextClassLoader().getResourc
e("").getPath();
String appConfigPath = rootPath + "app.properties";

Properties appProps = new Properties();
appProps.load(new FileInputStream(appConfigPath));

String appVersion = appProps.getProperty("version");
System.out.println("Version: "+ appVersion);
```

Exception Handling in Java

- Obligatory exception handling in Java → secure and reliable code
- Separation of concerns: business logic from exception handling code
- Class **Throwable** → classes Error и Exception
- Generating Exceptions – keyword **throw**
- Exception handling:
  - **try – catch – finally** block
  - Delegating the handling to the caller method - **throws**

Try-Catch-Finally Block

- Оператор **try** за изпълнение на несигурен код, множество **catch** блокове за обработка на изключения и **finally** за гарантиран clean-up накрая на обработката:

```
try {
   //код, който може да генерира изключения Ex1, Ex2, …
} catch(Ex1 ex) {  // изпълнява се само при Ex1
   //вземаме подходящи мерки за разрешаване на проблем
   1
} catch(Ex2 ex) {  // изпълнява се само при Ex2
    //вземаме подходящи мерки за разрешаване на
проблем 2
} finally {
    //изпълнява се винаги, независимо дали има
изключение
}
```

Exception Handling in Java - II

- Реализация на собствени изключения
- Конструктори с допълнителни аргументи
- Влагане и повторно генериране на изключения – причина Cause
- Специфика при обработката на **RuntimeException** и неговите наследници
- Завършване чрез **finally**

Novelties in Exception Handling since Java 7

- Multi-**catch** clause:

```
catch (Exception1|Exception2 ex) {

    ex.printStackTrace();

}
```

- Program block **try-with-resources**

```
String readInvoiceNumber(String myfile) throws IOException {
      try (BufferedReader input = new
            BufferedReader(new
              FileReader(myfile))) {
      return input.readLine();
        }
}
```

Enumeration Types

```
public class MyEnumeration {
  public enum InvoiceType { SIMPLE, VAT }
  public static void main(String[] args) {
    for(InvoiceType it : InvoiceType.values())
    System.out.println(it);
    }
}
```

Резултат: SIMPLE

VAT

Packages and Access Specifiers

- Packages and directories
- Importing packages – import
- Access specifiers
  - **public**
  - **private**
  - **protected**
  - Friendly access – by default within the package

Reusing Classes

- Advantages of code reuse
- Ways of implementation:
  - Objects composition
  - Inheritance of classes (object types)
- Building complex objects by composition
- Initializing the references:
  - on declaration of the site
  - in the constructor
  - before using (lazy initialization)

Class Inheritance - I

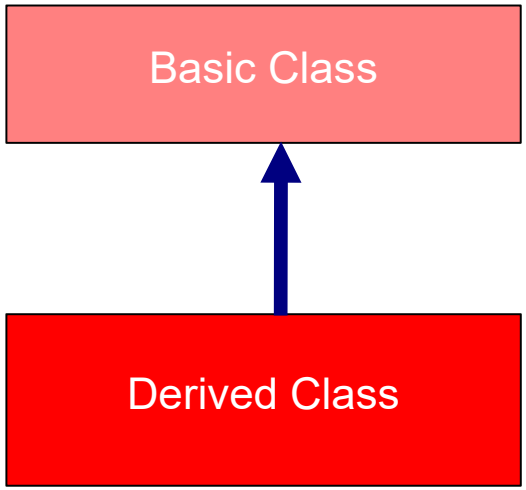
- Inheritance realization in Java™ language
  - Keyword **extends**
  - Keyword **super**
- Initialization of objects inheritance:
  1) base class; 2) inherited class
  - Calling the default constructors
  - Calling constructors with arguments
- Combining composition and inheritance

Class Inheritance - II

- Clearing of objects – realization in Java™
- Overloading and overriding methods of base class in derived classes
- When to use composition and when inheritance?
  - Do we need the interface of the base class?
  - Connection Type - „there is“ and „it is“?

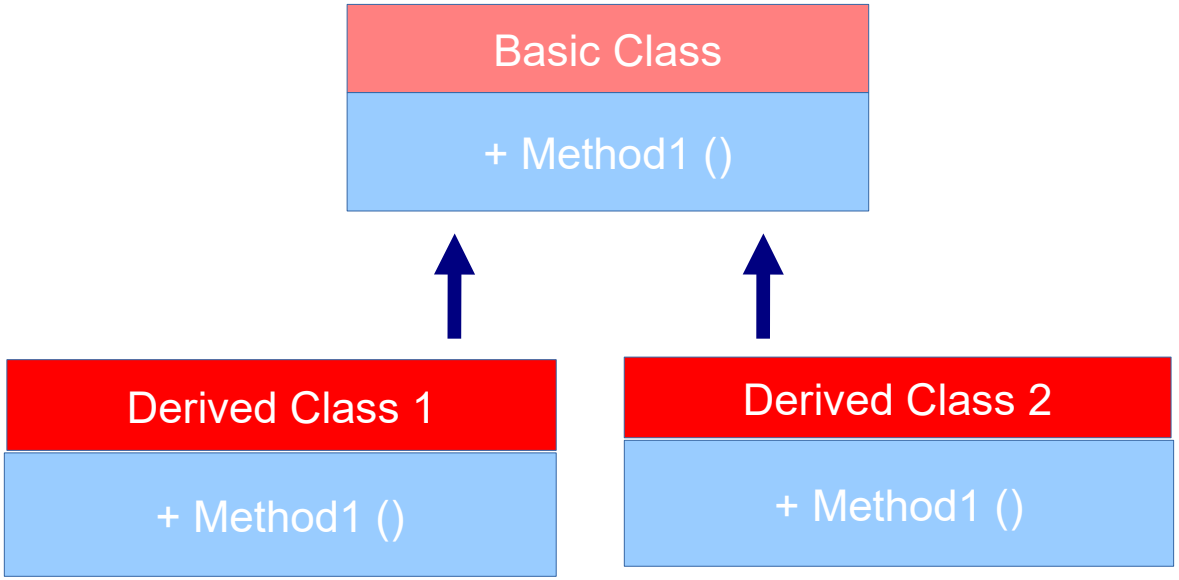
Class Inheritance - III

- Protected methods
- Upcasting
- Keyword final
  - Final data – defining constants
    - simple data type
    - objects
    - empty fields
    - arguments
  - Final methods
  - Final classes



Polymorphism - I

- Upcasting



- Abstract methods and classes – abstract
- Order of constructor calls
- Inheritance and expansion

Polymorphism - II

- Polymorphism – by default, unless the method is declared as static or final (private methods become automatically final)
- When constructing objects with inheritance each object cares about its attributes and delegate initialization of parental attributes on parental constructor or method
- Using polymorphic methods in constructor
- Covariance types of return (from Java SE 5)
- Composition <-> Inheritance - **State** Design Pattern

Interfaces and Multiple Inheritance

- Interfaces – keywords: interface, implements
- Multiple inheritance in Java
- Interface expansion through inheritance
- Constants (static final)
- Interface incorporation

Advantages of Using Interfaces

- Interfaces cleanly separate requirements type of the object from many possible implementations and make our code more universal and usable
- Reusable Design Pattern: Adapter – It allows to adapt existing realization interface that is required in our application
- Inheritance (expansion) of interfaces
- Reusable Design Pattern: Factory Method – creating reusable client code, isolated from the specifics of the particular server implementation

Inner Classes - I

- Inner Classes group logically related classes and control their visibility
- Closures – internal class has a constant connection to containing outside class and can access all its attributes and even final arguments and local variables (if defined in the method or block)
- Inner classes can be anonymous if used once in the program. Construction.
- Reference to the object from an external class - .this and creating an object from internal class in the context of containing object of the outer class - .new

Inner Classes - II

- Inner Classes
  - defined in an external class
  - defined in method
  - defined in a block of operators
  - access to the attributes of the outer class and to the arguments of the method which are defined in
- Anonymous inner classes
  - realizing public interface
  - inheriting class
  - instance initialization
  - static inner classes

Thank's for Your Attention!

Trayan Iliev

CEO of IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD

https://plus.google.com/+IproductOrg