



# CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS SPRING 2024

## LECTURE 2

Tamara Bonaci  
[t.bonaci@northeastern.edu](mailto:t.bonaci@northeastern.edu)

# ADMINISTRIVIA

---

- Lab 1 – due by 11:59pm on Thursday, January 18th
- HW1 – assigned today, due on Tuesday, January 23rd

# AGENDA

---

- Classes and objects
- Unit testing
- Object-Oriented Design - overview
- Inheritance and “is-a” relationship
  - Everything is an object
  - Equality (methods equals() and hashCode())
- Writing simple methods for classes
  - Methods for classes with containment
  - Methods that return objects
- Exceptions
- Documentation and Javadoc
- Visual representation and UML diagrams
- Codestyle

---

# CLASSES AND OBJECTS IN JAVA

CS 5004, SPRING 2024 – LECTURE 2

# OBJECTS AND CLASSES

---

- **Object** – an entity consisting of states and behavior
  - States stored in variables/fields
  - Behavior represented through methods
- **Class** – template/blueprint describing the states and the behavior that an object of that type supports

# OBJECT-ORIENTED DESIGN: THE BEGINNING

---

- **Classes** – templates/blueprints describing the states and behavior that an object of that type supports
- **Question: how do we design a class?**
- **Identify objects**
- **Identify properties**
- **Identify responsibilities**
- **Rule of thumb:**
  - Nouns – objects and properties
  - Verbs – responsibilities (methods)

# CLASSES AND VARIABLES IN JAVA

---

- **Classes** – templates/blueprints describing the states and behavior that an object of that type supports
- **Classes contain:**
  - **Local variables** – variables defined within any method, constructor or block
    - These variables are destroyed when the method has completed
  - **Instance variables** – variables within a class, but outside any method
    - Can be accessed from inside any method, constructor or blocks of a specific class
  - **Class variables** – variables declared within a class, outside of any method, with the keyword static

# ACCESS-CONTROL MODIFIERS IN JAVA

---

- In Java, there exist four access levels:
  - Visible to the package (default, no modifier needed)
  - Visible to the class only (modifier **private**)
  - Visible to the world (modifier **public**)
  - Visible to the package and all subclasses (modifier **protected**)



---

# UNIT TESTING

CS 5004, SPRING 2024 – LECTURE 2

# UNIT TESTING

---

- Unit testing - search for errors in a subsystem in isolation
  - A "subsystem" typically means a specific class or object
  - The Java library **JUnit** helps us to easily perform unit testing
- Basic idea:
  - For a given class Foo, create another class FooTest to test it, containing various "test case" methods to run
  - Each method looks for specific results, and either passes or fails
- JUnit provides "assert" commands to help us write tests
  - Idea - put assertion calls in your test methods to check things you expect to be true
  - If they are not, the test will fail

# JUNIT SETUP AND TEAR DOWN

---

Methods to run before/after each test case method is called:

@Before

```
public void name() { ... }
```

@After

```
public void name() { ... }
```

Methods to run once before/after the entire test class runs:

@BeforeClass

```
public static void name() { ... }
```

@AfterClass

```
public static void name() { ... }
```

# JUNIT TESTING - EXAMPLE

---

```
/**
 * Simple class Person, that includes private instance variables firstName and lastName.
 */
public class Person {

    private Name personsName;
    private String address;

    public Person(Name personsName, String address) {
        this.personsName = personsName;
        this.address = address;
    }

    public void setPersonsName(Name personsName) {
        this.personsName = personsName;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public Name getPersonsName() { return personsName; }

    public String getAddress() { return address; }
}
```

# JUNIT TESTING - EXAMPLE

---

```
import junit.framework.TestCase;
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class PersonTest {
    Person testPerson;
    String expectedName = "John Doe";
    String expectedEmail = "john_doe12345@gmail.com";
    String expectedAddress = "N/A";

    @Before
    public void setUp() throws Exception {
        testPerson = new Person( personsName: "John Doe", address: "john_doe12345@gmail.com", "N/A");
    }

    @Test
    public void getName() {
        TestCase.assertEquals(this.expectedName, testPerson.getName());
    }
}
```

---

# OBJECT-ORIENTED DESIGN - OVERVIEW

CS 5004, SPRING 2024 – LECTURE 2

# OBJECT-ORIENTED DESIGN PRINCIPLES

---

- Inheritance
- Abstraction
- Encapsulation
- Information hiding
- Polymorphism

# OBJECT-ORIENTED DESIGN PRINCIPLES: INHERITANCE

---



[Pictures credit: <https://medium.com/java-for-absolute-dummies/inheritance-in-java-programming-39176e0016f3>]

**Inheritance** – an ability for a class to extend or override functionality (states and behavior) of other classes



# OBJECT-ORIENTED DESIGN PRINCIPLES: ABSTRACTION

---



[Pictures credit:<https://www.pinterest.com/pin/423408802450668522/>]

**Abstraction** - an ability to segregate implementation from an interface

# OBJECT-ORIENTED DESIGN PRINCIPLES: ENCAPSULATION

---



[Pictures credit:<http://small-pets.lovetoknow.com/reptiles-amphibians/names-pet-turtles>]

**Encapsulation idea** – data types and methods operating on that data coupled within an object/class

# OBJECT-ORIENTED DESIGN PRINCIPLES: INFORMATION HIDING

---

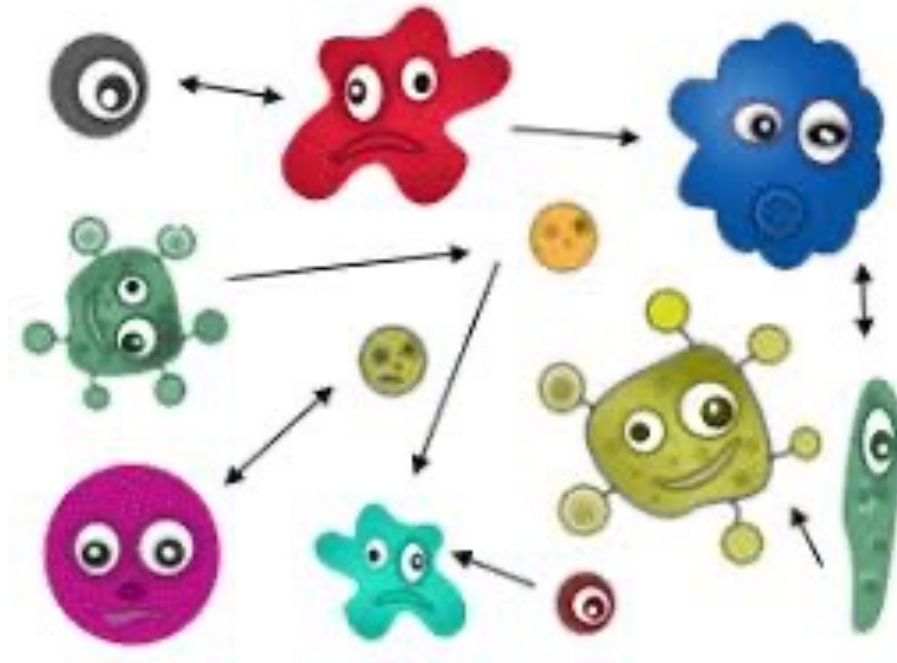


[Pictures credit:<http://www.sickchirpse.com/kebab-shop-owner-three-days-hiding-london-terror-attacks/>]

**Information hiding idea** – expose only the necessary functionality (through interface), and hide everything else

# OBJECT-ORIENTED DESIGN PRINCIPLES: POLYMORPHISM

---



[Pictures credit: <http://www.thewindowsclub.com/polymorphic-virus>]

**Polymorphism** – the ability to define different classes and methods as having the same name but taking different data types

---

# INHERITANCE AND “IS A” RELATIONSHIP

CS 5004, SPRING 2024 – LECTURE 2

# INHERITANCE AND “IS A” RELATIONSHIP

---

- **Inheritance** - set of classes connected by an ‘is-a’ relationship
- **‘Is-a’ relationship** - hierarchical connection where one category can be treated as a specialized version of another
  - **Example 1:**
    - Every student is a person
    - Every ALIGN student is a student
  - **Example 2:**
    - Every pepper is a vegetable
    - Every bell pepper is a pepper
    - Every banana pepper is a pepper

# CLASS INHERITANCE

---

- Many programming languages (Java, C++, C#) provide a direct support for is-a relationship through class inheritance
- Class inheritance - new class extends existing class
  - Original/Extended class (also known as base class or super class)
  - New/Extending class (also known as derived class or subclass)
- Rules for derived classes (subclasses):
  - Derived class automatically inherits all NON-private instance variables and methods of the base class
  - Derived class can add additional methods and instance variables
  - Derived class can provide different versions of inherited methods
- **Note:** in Java, a class can extend **only** one class

# CLASS INHERITANCE

Derived class automatically inherits all NON-private instance variables and methods of the base class


```
public class Person {  
    protected String firstName;  
    protected String lastName;  
}
```

Parent class with  
protected fields



```
public class Student extends Person {  
    private String studentID;  
  
    public Student(String firstName, String lastName, String studentID) {  
        super(firstName, lastName);  
        this.studentID = studentID;  
    }  
}
```

Child class – direct access  
to the parent's protected  
fields





# CLASS INHERITANCE

Derived class automatically inherits all NON-private instance variables and methods of the base class

```
public class Person {  
    private String firstName;  
    private String lastName;  
}
```

← Parent class with  
private fields

```
public class Client extends Person {  
    private String clientID;  
  
    public Client(String firstName, String lastName, String studentID) {  
        super(firstName, lastName);  
        this.clientID = studentID;  
    }  
}
```

← Child class – no direct  
access to the parent's  
protected fields

# EVERYTHING IS AN OBJECT IN JAVA

---

- `public class Object` - the root of the class hierarchy
  - Every class has `Object` as a superclass
  - All objects inherit public methods of `Object`

|  |  |
|--|--|
| <code>protected Object clone()</code>    | Creates and returns a copy of this object.   |
| <code>Boolean equals (Object obj)</code> | Indicates whether some other object is "equal to" this one.  |
| <code>protected void finalize()</code>   | Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. |
| <code>Class&lt;?&gt; getClass()</code>   | Returns the runtime class of this <code>Object</code> .  |
| <code>int hashCode()</code>              | Returns a hash code value for the object.  |
| <code>void notify()</code>               | Wakes up a single thread that is waiting on this object's monitor.   |
| <code>String toString()</code>           | Returns a string representation of the object.   |

---

# EQUALITY

CS 5004, SPRING 2024 – LECTURE 2

# EQUALITY OF OBJECTS

---

How do we compare objects?

# CLASS OBJECT – ROOT OF THE CLASS HIERARCHY

## Constructor Summary

### Constructors

#### Constructor and Description

`Object()`

## Method Summary

### Methods

| Modifier and Type             | Method and Description  |
|-------------------------------|---|
| protected <code>Object</code> | <code>clone()</code><br>Creates and returns a copy of this object.  |
| boolean                       | <code>equals(Object obj)</code><br>Indicates whether some other object is "equal to" this one.  |
| protected void                | <code>finalize()</code><br>Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.   |
| <code>Class&lt;?&gt;</code>   | <code>getClass()</code><br>Returns the runtime class of this object.  |
| int                           | <code>hashCode()</code><br>Returns a hash code value for the object.  |
| void                          | <code>notify()</code><br>Wakes up a single thread that is waiting on this object's monitor.   |
| void                          | <code>notifyAll()</code><br>Wakes up all threads that are waiting on this object's monitor.   |
| <code>String</code>           | <code>toString()</code><br>Returns a string representation of the object.   |
| void                          | <code>wait()</code><br>Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object.  |
| void                          | <code>wait(long timeout)</code><br>Causes the current thread to wait until either another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or a specified amount of time has elapsed.  |
| void                          | <code>wait(long timeout, int nanos)</code><br>Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed. |

# EXPECTED PROPERTIES OF EQUALITY

---

- **Reflexive:** `a.equals(a) == true`
  - Confusing if an object does not equal itself
- **Symmetric:** `a.equals(b)  $\leftrightarrow$  b.equals(a)`
  - Confusing if order-of-arguments matters
- **Transitive:** `a.equals(b) && b.equals(c)  $\rightarrow$  a.equals(c)`
  - Confusing again to violate centuries of logical reasoning
    - A relation that is reflexive, transitive, and symmetric is an *equivalence relation*

# SPECIFICATION FOR METHOD EQUALS()

---

- `public boolean equals(Object obj)`- indicates whether some other object is “equal to” this one.
- The equals method implements an **equivalence relation**:
  - **It is reflexive**: for any reference value `x`, `x.equals(x)`
  - **It is symmetric**: for any reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
  - **It is transitive**: for any reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
- **It is consistent**: for any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
- For any non-null reference value `x`, `x.equals(null)` should return false

# METHOD HASHCODE()

---

- Another method in Object- `public int hashCode()`
- Returns a hash code value for the object
- This method is supported for the benefit of hash tables such as those provided by `java.util.HashMap`.”
- Contract (again essential for correct overriding):
  - Self-consistent:
    - `o.hashCode() == o.hashCode()`
    - -..so long as o doesn't change between the calls
  - Consistent with equality:
    - `a.equals(b) → a.hashCode() == b.hashCode()`



# IDEA: THINK OF HASHCODE() AS A PRE-FILTER

---

- If two objects are equal, they **must** have the same hash code
  - Up to implementers of methods hashCode() and equals() to achieve that
  - **If you override equals(), you must override hashCode() too**
- If two objects have the same hash code, they still may or may not be equal
  - “Usually not” leads to better performance
  - hashCode() in Object tries to (but may not) give every object a different hash code
- Hash codes are usually cheaper to compute, so check first if you “usually expect not equal” – **a pre-filter**

---

# EXCEPTIONS

CS 5004, SPRING 2024 – LECTURE 2

# WHY EXCEPTIONS?

---

- **Example:** class CheckingAccount, with methods:
  - deposit()
  - withdraw()
- Code example in the course code
- **Question:** is there something that could go wrong with these methods?

# EXCEPTIONS

---

- **Exception** occurs when something unexpected happens during program execution:
  - Invalid input provided
  - Operation cannot be completed (e.g., square root of a negative number)
  - Something beyond program control happens (e.g. attempting to read a file that no longer exists)
- **Exceptions provide a graceful way of aborting the method and sending the message to its caller that something went wrong**

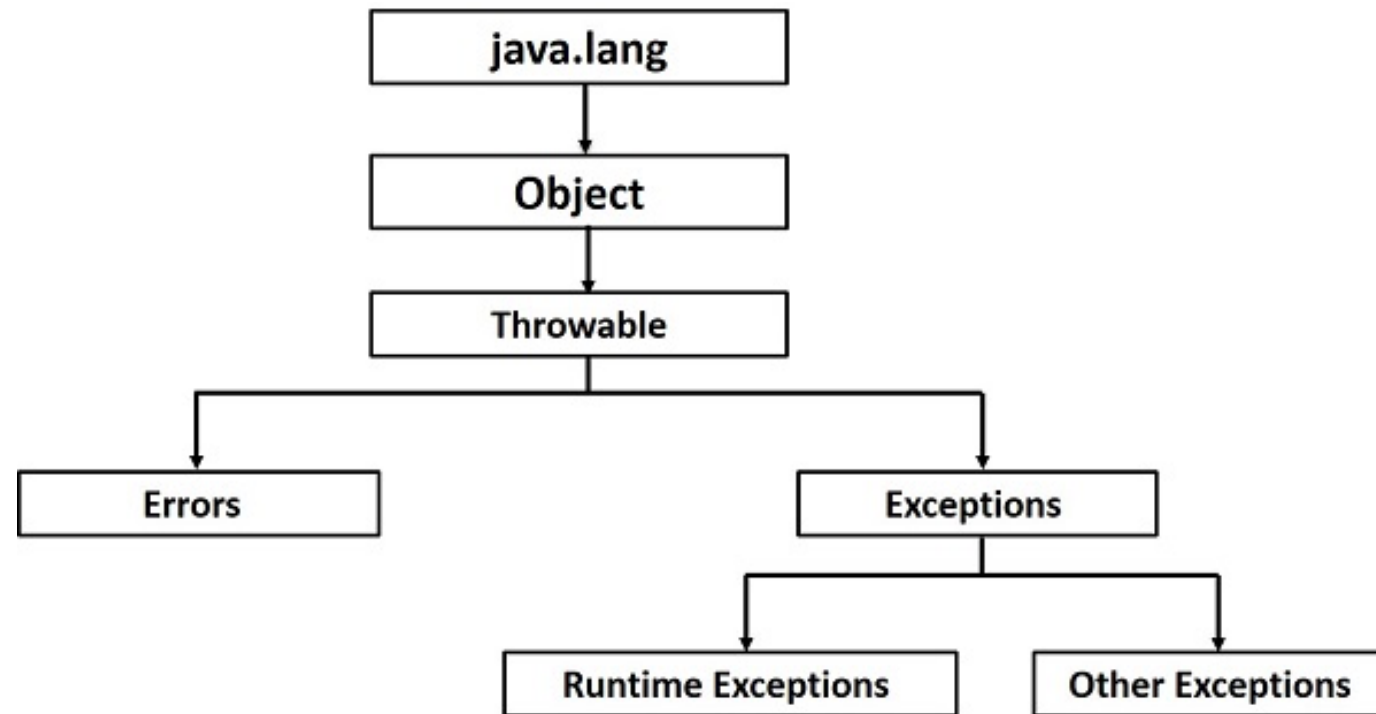
# TYPES OF EXCEPTION

---

- **Exception** occurs when something unexpected happens during the program execution
- **Types of exceptions:**
  - **Checked exception** - an exception that occurs at the compile time (compile time exceptions)
  - **Unchecked exception** - an exception that occurs at the time of execution (runtime exceptions)
- **Errors** – errors are not exceptions, but problems that arise beyond the control of the user or the programmer

# EXCEPTIONS IN JAVA - HIERARCHY

- All exception classes in Java are subtypes of the java.lang.Exception class



[Figure credit:[https://www.tutorialspoint.com/java/java\\_exceptions.htm](https://www.tutorialspoint.com/java/java_exceptions.htm)]

# USING EXCEPTIONS – A GENERAL RECIPE

---

1. Design/write a method signature
2. Write the method body so that it works correctly under ideal circumstances
3. Carefully think about all non-ideal situations may can occur, other than ideal circumstances
4. For each issue that may error:
  - If there is a way to prevent it from happening, do it (the absolute best way of error handling)
  - If there isn't a way to prevent the error, but there is a way to recover from it, do it (there is no need to use exceptions, as the recovery happens in the same method as the error)
  - Otherwise, choose an appropriate exception type for the error:
    - Declare that this method throws this exception
    - Throw this exception appropriately

# WRITING A METHOD WITH AN EXCEPTION

---

```
/**
 * Compute and return the price of this book with the given discount (as a
 * percentage)
 *
 * @param discount the percentage discount to be applied
 * @return the discounted price of this book
 * @throws IllegalArgumentException if a negative discount is passed as an
 * argument
 */
public float salePrice(float discount) throws IllegalArgumentException {
    /* TEMPLATE:
     * this.price: float
     * this.author: Person
     * this.title: String
     *
     * Parameters:
     * discount: float
     */
    if (discount < 0) {
        throw new IllegalArgumentException("Discount cannot be negative");
    }

    return this.price - (this.price * discount) / 100;
}
```

*In method signature, declare that  
an exception can be thrown*

*Throw an exception*



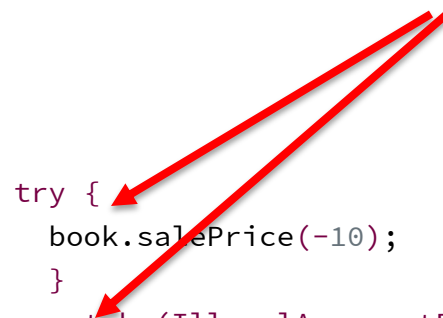
# CALLING A METHOD WITH AN EXCEPTION

---

Try-catch block:

Start by executing try block

If exception `IllegalArgumentException` caught,  
execute code in the catch block



```
try {  
    book.salePrice(-10);  
}  
catch (IllegalArgumentException e) {  
    //This will be executed only if an IllegalArgumentException is thrown by the above method call  
}
```

# CATCHING AN EXCEPTION

---

- Catching a single exception:

```
try {  
    // Protected code  
}  
catch (ExceptionName e1) {  
    // Catch block  
}
```

- Catching multiple exceptions:

```
try {  
    // Protected code  
}  
catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
}
```

# CATCHING AN EXCEPTION – THE FINALLY BLOCK

---

```
try {  
    // Protected code  
} catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
}finally {  
    // The finally block always executes.  
}
```

# TESTING A METHOD WITH AN EXCEPTION

---

- When testing a method that can throw an exception, there are several things to check:
  - That the method is functionally correct within its valid operating range (happy path)
  - That the method does not throw an exception when it is not supposed to (if it does, a test fails)
  - That the method throws an exception when expected (if it doesn't, a test fails)

# WRITING YOUR OWN EXCEPTIONS

---

- You can create your own exceptions in Java
- Some rules:
  - All exceptions inherit the behavior of class Throwable
  - If you want to write a checked exception, extend class Exception
  - If you want to write a runtime exception, extend class RuntimeException

---

# DOCUMENTATION AND JAVADOC

CS 5004, SPRING 2024 – LECTURE 2

# COMMENTING AND DOCUMENTATION

---

- **Documentation** - comments explaining your design and purpose
- **General documentation rules:**
  - **Classes** - explain in a sentence or two what a class represents
    1. **Semantic details** - what the class represents from the problem statement point of view
    2. **Technical details** - things that you would like to know before, or in order to interact with the class
    3. Any and all details that a user of this class may need to know to use it properly
  - **Methods** - the purpose statement for the method
    1. List all arguments and what they represent
    2. Explain what the method returns
    3. **Include documentation for constructor and getters too**
  - **Within the method body** - mention any details that you think are relevant to what that method is doing

# JAVADOC

---

- **Javadoc** - documentation generator for Java
- Generates API documentation in the HTML format from the Java source code and included comments
- **Structure of a Javadoc comment:**
  - A Javadoc comment differs from a regular code by standard multi-line comment tags `/*` and `*/`
  - The opening tag (a.k.a. **the begin-comment delimiter**), has an extra asterisk, as in `/**`
  - The first paragraph is a description of a class/method being documented
  - **Following the description are a varying number of descriptive tags:**
    - The parameters of the method (`@param`)
    - What the method returns (`@return`)
    - Any exceptions the method may throw (`@throws`)
    - Other less-common tags such as `@see` (a "see also" tag)



# STRUCTURE OF A JAVADOC

---

- A Javadoc must precede a class, a field, a constructor, a method declaration
- Made up of two parts:
- Description of tags, ordered as:
  - `@author` (classes and interfaces only, required)
  - `@version` (classes and interfaces only, required)
  - `@param` (methods and constructors only)
  - `@return` (methods only)
  - `@throws`
  - `@see`
  - `@since`
  - `@serial` (or `@serialField` or `@serialData`)
  - `@deprecated` (see [How and When To Deprecate APIs](#))

# STRUCTURE OF A JAVADOC

---

- A Javadoc must precede a class, a field, a constructor, a method declaration
- Required tags:
  - `@param` tag is **required** (by convention) for every parameter, even when the description is obvious
  - `@return` tag is required for every method that returns something other than void, even if it is redundant with the method description

# JAVADOC COMMENT - EXAMPLE

```
/**
 * Starting class Author, copied from the lab assignment.
 *
 */
public class Author {

    private Name name;
    private String email;
    private String address;

    /**
     * Creates a new author given the author's name, email and address as strings.
     *
     * @param name the author's name
     * @param email the author's email address
     * @param address the authors physical address
     */
    public Author(Name name, String email, String address) {
        this.name = name;
        this.email = email;
        this.address = address;
    }

    /**
     * @return the name
     */
    public Name getName() { return this.name; }

    /**
     * @return the email
     */
    public String getEmail() { return this.email; }

    /**
     * @return the address
     */
}
```

# JAVADOC COMMENT - EXAMPLE

## Class Author

java.lang.Object  
Author

```
public class Author
extends java.lang.Object
```

Starting class Author, copied from the lab assignment.

### Constructor Summary

#### Constructors

| Constructor   | Description   |
|---|---|
| <b>Author</b> (Name name, java.lang.String email, java.lang.String address) | Creates a new author given the author's name, email and address as strings. |

### Method Summary

#### All Methods Instance Methods Concrete Methods

| Modifier and Type | Method               | Description |
|-------------------|----------------------|-------------|
| java.lang.String  | <b>getAddress</b> () |             |
| java.lang.String  | <b>getEmail</b> ()   |             |
| Name              | <b>getName</b> ()    |             |

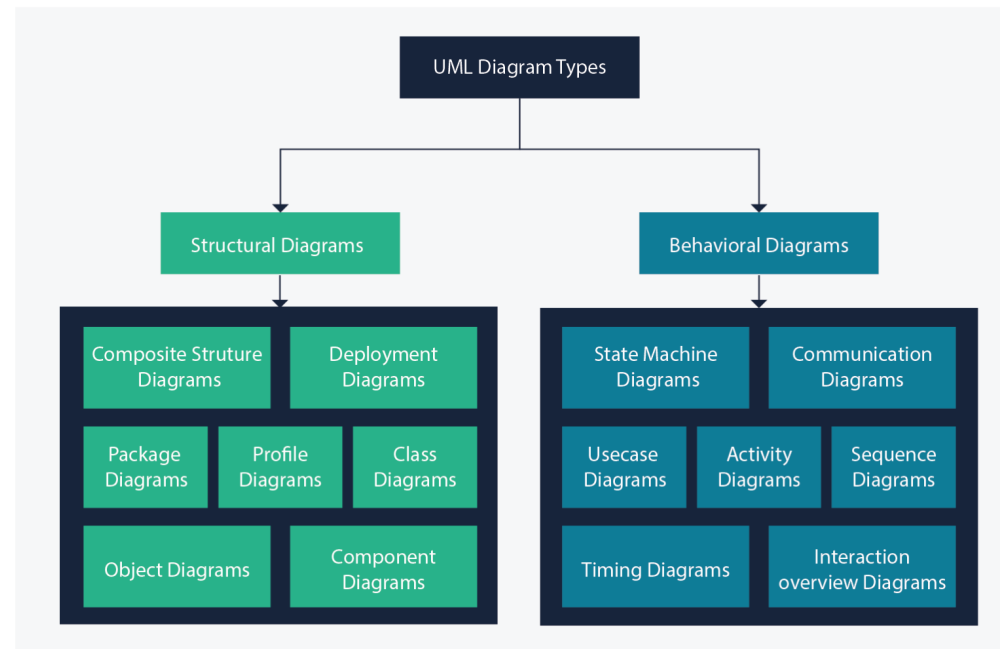
---

# **VISUAL REPRESENTATIONS AND UML DIAGRAMS**

CS 5004, SPRING 2024 – LECTURE 2

# UML DIAGRAMS

- **UML (Unified Modeling Language)** – language used to model software solutions, applications structures, system behavior and business processes
- Two main categories of UML diagrams: **structure** and **behavior diagrams**



[Figure credit: <https://creately.com/blog/diagrams/uml-diagram-types-examples/>]

# CLASS UML DIAGRAM

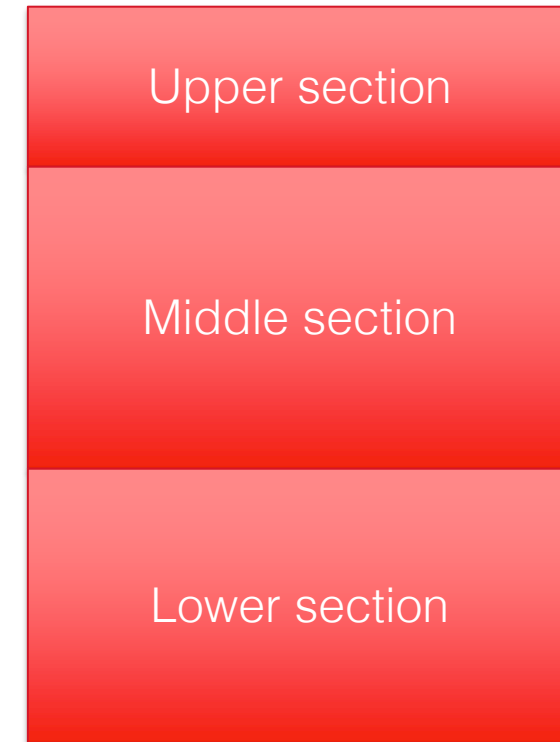
---

- **Class UML diagram** – the main building block of any object-oriented solution, showing:
  - The classes involved in the solution
  - **Relationships** between those classes
  - **Fields** (states/attributes) of every class
  - **Methods** (operations/behavior) of every class
- **Benefits of Class UML diagrams:**
  - Allow for a better understanding of a problem at hands through a graphical representation
  - Provide an implementation-independent description of data types (objects) that are used in a system, and are later passed between its components
  - Provide a detailed representation of a specific code needed to be programmed and implemented for a specific solution

# BASIC COMPONENTS OF A CLASS UML DIAGRAM

---

- **Upper section** - contains the name of the class (required section)
- **Middle section** - contains the fields (attributes)
  - This section is only required when describing a specific instance of a class
- **Bottom section** - includes class methods (operations)
  - Each operation takes up its own line





# BASIC COMPONENTS OF A CLASS UML DIAGRAM

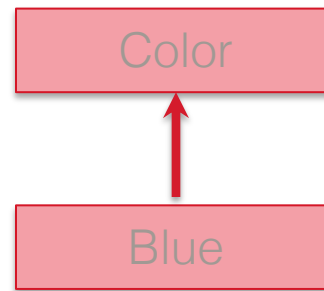
---

- Classes can have different access levels, depending on their access modifiers (visibility)
- **Corresponding UML symbols for various access levels:**
  - `public (+)`
  - `private (-)`
  - `protected (#)`
  - `package (~)`
  - `derived (/)`
  - `static (underlined)`

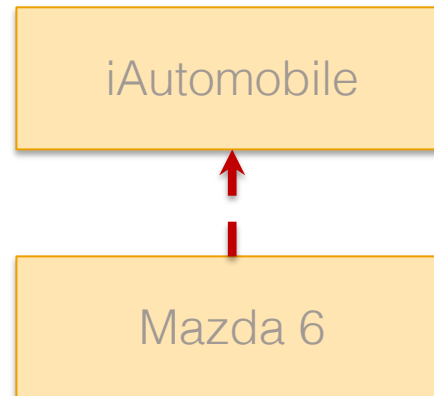
# CLASS UML DIAGRAM - RELATIONSHIPS

---

- **Inheritance/generalization** - a type of relationship wherein one associated class is a child of another, by virtue of assuming the same functionalities of the parent class (the child class is a specific type of the parent class)



- **Realization** - denotes the implementation of the functionality defined in one class by another class



# CLASS UML DIAGRAM - RELATIONSHIPS

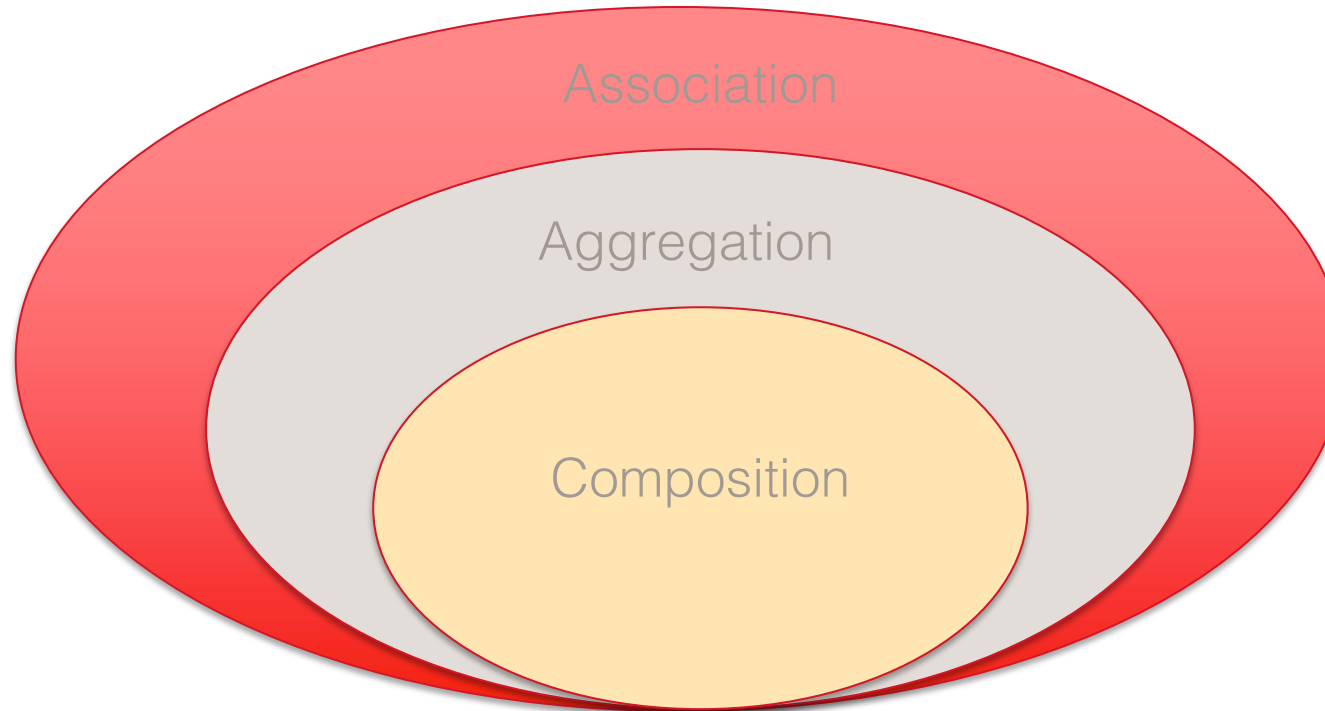
---

- **Association** - a broad term that encompasses just about any logical relationship between classes
- **Direct association** - a directional relationship, represented by a line with an arrowhead where an arrowhead depicts a container-contained directional flow
- **Aggregation** – a relationship where one class contains another class (is aggregated out of) instances of another class
  - The contained classes are not strongly dependent on the lifecycle of the container
  - **Example** - some class Library is made up of one or more objects Book
    - If the Library is dissolved, objects Books may still remain alive
- **Composition** – another relationship where one class contains instances of another class, but such that the dependence of the contained class to the life cycle of the container class is **emphasized** (the contained class will be obliterated when the container class is destroyed)
  - **Example** - a ShoulderBag's SidePocket will also cease to exist once the shoulder bag is destroyed

# CLASS UML DIAGRAM - RELATIONSHIPS

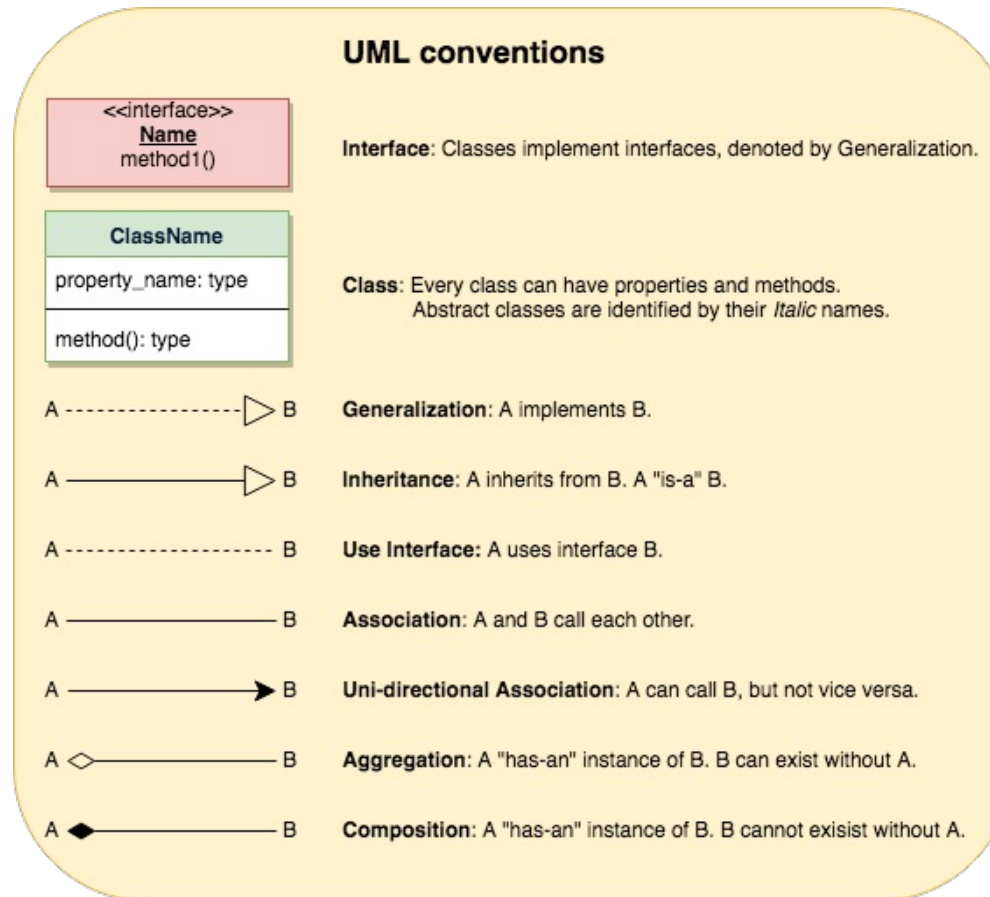
---

- Association, Aggregations and Composition



- TL;DR – when in doubt between aggregation and composition, check aggregation first

# CLASS UML DIAGRAM - SUMMARY



[Figure credit: [www.education.io](http://www.education.io)]

---

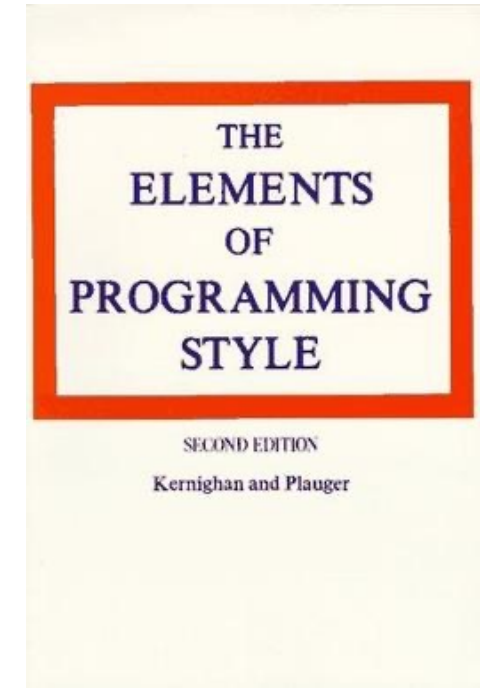
# **CODESTYLE**

CS 5004, SPRING 2024 – LECTURE 2

# CODE (PROGRAMMING) STYLE

---

- **Code (programming) style** - set of rules used when writing code, in order to increase understanding and readability of the code
- It started in the 1970-ies with the **Elements of Programming Style**

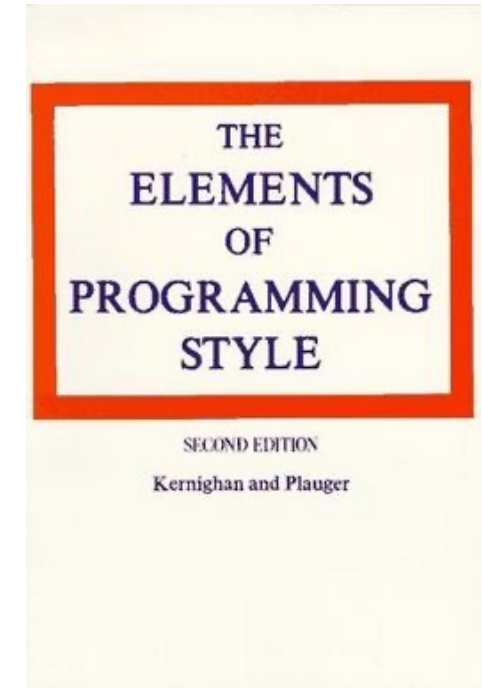


[Picture credit: wikipedia.com]

# CODE (PROGRAMMING) STYLE

---

- Code style usually defines the layout of the source code:
  - Indentation
  - Use of white space around operator and keywords
  - Capitalization or not of keywords and variable names
  - Style and spelling of user-defined identifiers (classes, methods, variables)
  - Use and style of comments



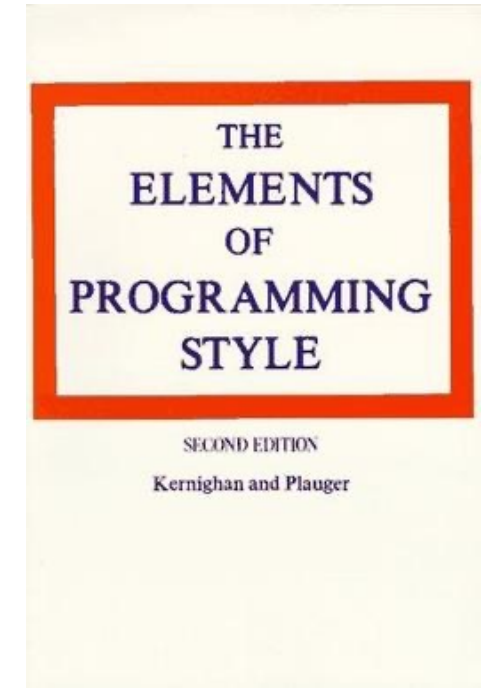
[Picture credit: wikipedia.com]



# CODE (PROGRAMMING) STYLE

---

- We will use Google Java Code Style:
- <https://google.github.io/styleguide/javaguide.html>
- <https://github.com/google/styleguide/blob/gh-pages/intelij-java-google-style.xml>
- IntelliJ: Preferences → Editor → Code Style → Java → manage



[Picture credit: wikipedia.com]

---

# IMMUTABILITY

CS 5004, SPRING 2024 – LECTURE 2

# IMMUTABLE OBJECTS

---

- **Immutable object** - an object whose internal state remains constant after it has been entirely created

# STRATEGIES FOR DEFINING IMMUTABLE OBJECTS

---

[Resource credit: <https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html>]

- Make all fields final and private
- Don't provide setter methods
- Don't allow subclasses to override methods
  - The simplest way to do this is to declare the class as final
  - Another approach is to make the constructor private and construct instances in factory methods
- If the instance fields include references to mutable objects, don't allow those objects to be changed:
  - Don't provide methods that modify the mutable objects
  - Don't share references to the mutable objects
  - Never store references to external, mutable objects passed to the constructor
  - Create copies of your internal mutable objects when necessary to avoid returning the originals in your methods

# STRATEGIES FOR DEFINING IMMUTABLE OBJECTS

---

[Resource credit: <https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html>]

- **Note1:** the presented rules define a simple strategy for creating immutable objects, but not all classes documented as "immutable" follow these rules
  - This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction
- *Note 2: we don't follow all of these rules, and we won't enforce them, but we wanted you to be aware of them*

# ADVANTAGES OF IMMUTABLE OBJECTS

---

- **Thread-safety:** immutable object remains in exactly one state, the state in which it was created → the object is thread-safe
- **Unique elements of data collections:** since immutable objects do not change value after creation, we only need one instance of every object → unique objects, that can easily be used in Maps and Sets
- **Easily established and maintained class invariance:** with immutable objects, class invariance is established once, and then unchanged
- **No conflicts among objects:** immutability makes it easier to parallelize program as there are no conflicts among objects
- **Consistent internal state of program even with exceptions**
- References to immutable objects can be cached as they are not going to change

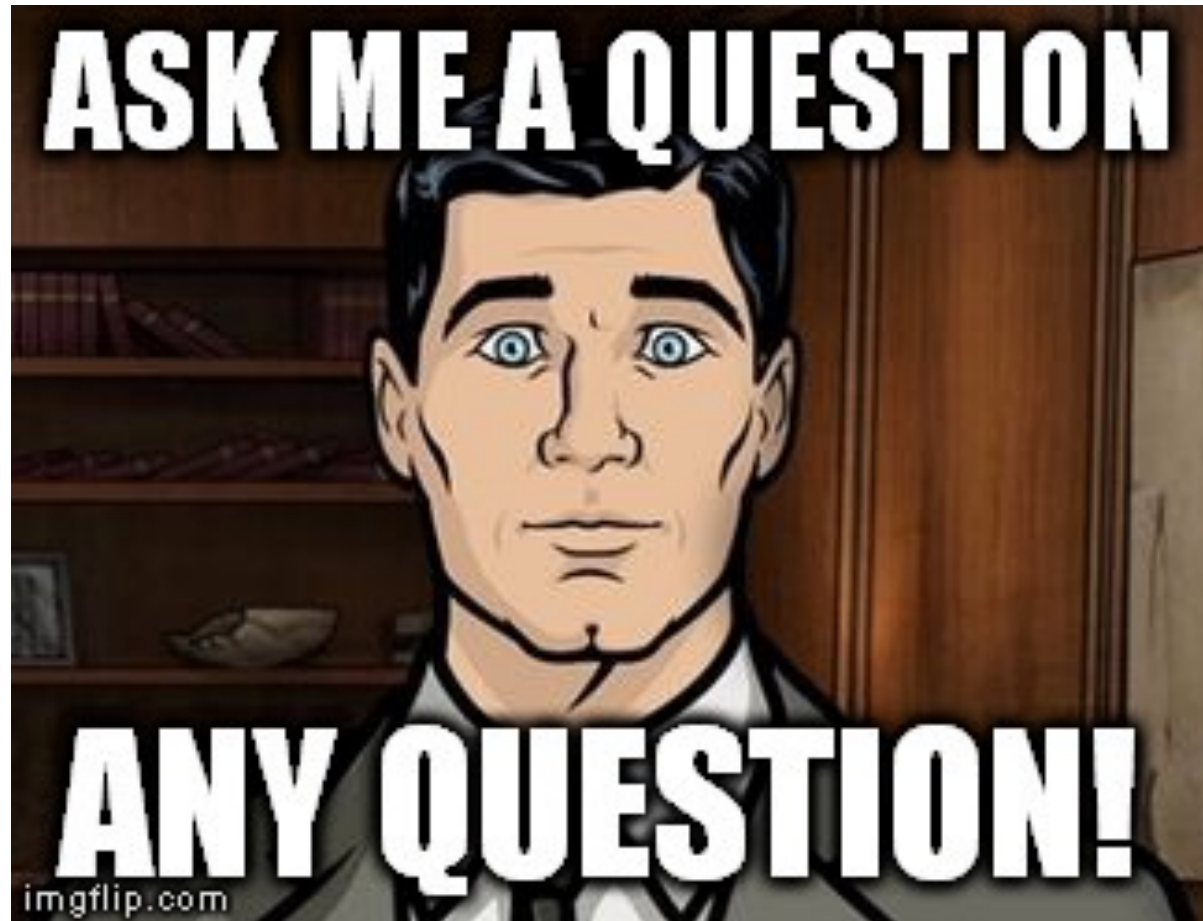
# DISADVANTAGES OF IMMUTABLE OBJECTS

---

- Whenever an object needs to be modified → a new object needs to be created, potentially causing more frequent garbage collections

# YOUR QUESTIONS

---



[Meme credit: imgflip.com]



# REFERENCES AND READING MATERIAL

---

- Java Getting Started (<https://docs.oracle.com/javase/tutorial/getStarted/index.html>)
- Object-Oriented Programming Concepts (<https://docs.oracle.com/javase/tutorial/java/concepts/index.html>)
- Language Basics (<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html>)
- How to Design Classes (HtDC), Chapters 1-3
- JUnit: Getting Started (<https://github.com/junit-team/junit4/wiki/Getting-started>)
- JUnit: Assertions (<https://github.com/junit-team/junit4/wiki/Assertions>)
- Unit testing with JUnit: <http://www.vogella.com/tutorials/JUnit/article.html>
- Java Tutorial: Interfaces and Inheritance: <https://docs.oracle.com/javase/tutorial/java/landl/index.html>
- Java – Exceptions ([https://www.tutorialspoint.com/java/java\\_exceptions.htm](https://www.tutorialspoint.com/java/java_exceptions.htm))
- Declare Your Own Exception ([https://www.ibm.com/developerworks/community/blogs/738b7897-cd38-4f24-9f05-48dd69116837/entry/declare\\_your\\_own\\_java\\_exceptions?lang=en](https://www.ibm.com/developerworks/community/blogs/738b7897-cd38-4f24-9f05-48dd69116837/entry/declare_your_own_java_exceptions?lang=en))