Northeastern University

CS 5010: Programming Design Paradigms
Fall 2022

Lecture 2: Whirlwind Tour of Java

Brian Cross
b.cross@northeastern.edu

Administrivia

- First assignment due on **Monday, September 26th by 12pm NOON**
  - UML Predesign due Wednesday, 21st, by 10am
- Lab 2 on Monday, September 19th
- Sample Hello Java code: https://github.ccs.neu.edu/cs5010seaF22/Code_From_Lectures

Administrivia

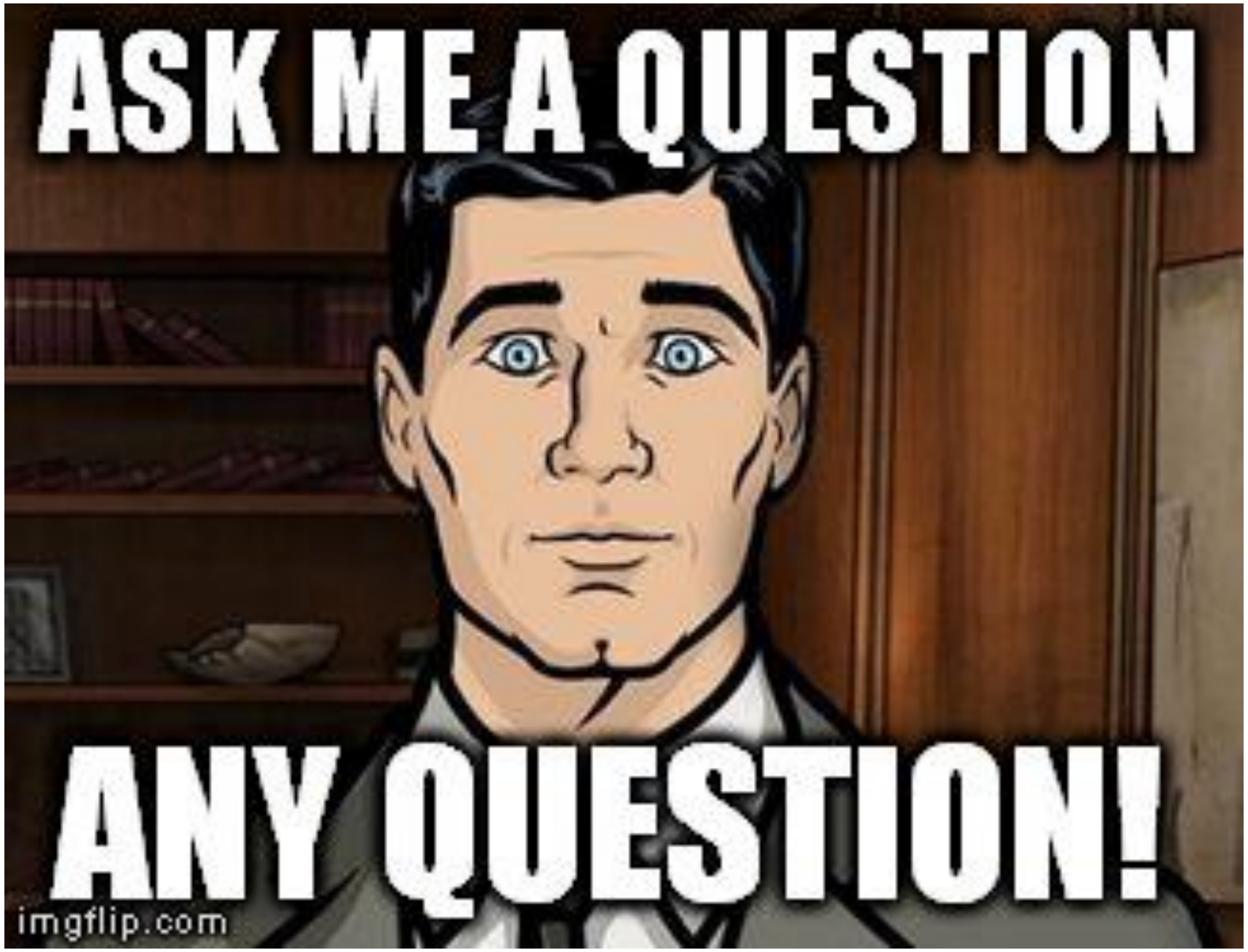
- How to submit your labs and homework?
    - Using Khoury GitHub
    - Create Pull Request, do not merge to **main**
    - Lab 2 on Monday – Topic is Git
- By now, you should have:
    1. Setup your Khoury account
    2. Accessed Piazza
    3. Answered Survey with Khoury username
    4. Plan to access your Khoury GitHub at least once before next lab
- If you haven't done this, or you don't have access to your GitHub repo, let us know ASAP

Administrivia

- What to do if I have questions?
  1. General question – post it **publicly** on Piazza
  2. Assignment specific question/sensitive question
     - Send a private Piazza message to all **instructors**, or
     - Come to office hours

- Office Hours – still TBD (Early next week)
  - Survey data

Your Questions



[Meme credit: imgflip.com]

Agenda – Whirlwind Tour of Java

- Design By Contract
- Introduction to Java
- Objects and classes in Java
- Data Types in Java
- Modifier Types in Java
- Strings in Java
- Equality In Java
- Testing Code
- Exceptions
- UML
- **Documentation: Javadoc**
- Code Style

DESIGN BY Contract

The Ripple Effect

- A seemingly simple change leads to many unexpected changes
- Parts of the programs are dependent upon each other
- Change one, must change many – tightly coupled
- The number of interactions/dependencies makes code unmanageable

Modularity

- Decompose the problem into parts
  - Modules, packages, classes, components, etc.
- Create minimal dependencies between the parts
  - Loosely coupled, limit ripple effect
- Dependencies based on specifications
  - Hide implementation details from other parts
  - Details can change as long as specification not violated

Specification

- Defines a contract between a ‘using’ class and a ‘used’ class
  - e.g client, server
- Describes expectations of each other
  - What data the client must pass to the server?
  - What effects passing the expected data will have on the server?
  - What the server will return to the client?
  - What conditions can be guaranteed to hold after the request is complete?

Elements of a Contract

- Preconditions of the module
    - What conditions the module requests from its clients
    - Check upon entry to module
- Postconditions of the module
    - What guarantees the module gives to clients
    - What conditions must hold for all objects of this module if implemented correctly

When to Check?

- Preconditions
  - Upon module entry
  - Or as early as feasible
    - Throw an exception if violated
- Postconditions
  - Just before returning
  - Violations indicate errors in the module
    - Useful for debugging
    - In production?

Whirlwind Tour of Java

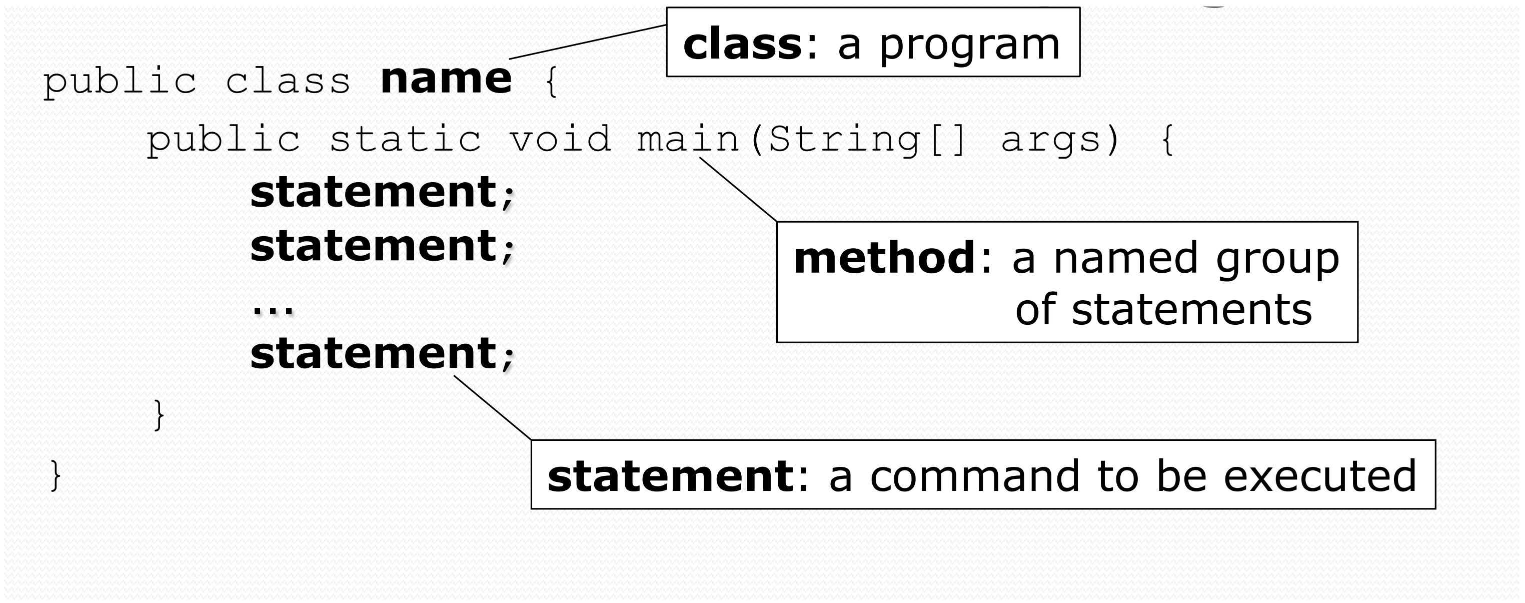
INTRODUCTION TO JAVA

Java – A Popular Programming Language

- Object-oriented
- Many well-developed IDEs
- Tons of pre-written software (not all of it good ☺)
- Platform independent
  - Java compiler compiles the source code into byte code which runs on many different computer types

Structure of a Java Program

- Every executable Java program consists of a class,
  - That contains a method named `main`
    - That contains statements (commands) to be executed



```
public class name {
     public static void main(String[] args) {
          statement;
          statement;
          ...
          statement;
     }
}
```

class: a program
method: a named group of statements
statement: a command to be executed

[Graphic credit: Paerson Education]

Our First Java Program

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, PDP_Fall_2022!");
        System.out.println();
        System.out.println("Keep up the good work");
    }
}
```

Output?

- System.out.println – two signatures
  - Print a message: System.out.println(“Hi!”);
  - Print a blank line: System.out.println();

Whirlwind Tour of Java

OBJECTS AND CLASSES IN JAVA

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

Classes 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 that particular class*
  - Class variables – variables declared within a class, outside of any method, with the keyword **`static`**

Classes and Constructors in Java

- Classes – templates/blueprints describing the states and behavior that an object of that type supports

- Every class has a constructor
  - *In Java, if we don't explicitly write a constructor, Java compiler builds a default constructor for that class*
  - *But it is a good practice to write a constructor, even an empty one*

Creating an Object in Java

- In Java, an object is created from a class using the keyword **new**

- Three steps involved when creating an object from a class:
  1. Declaration – a new variable is declared, with a variable name, and object type
  2. Instantiation – an object is created using the keyword **new**
  3. Initialization – an object is initialized using the keyword **new** + a constructor call

Creating an Object in Java

- Example: creating an object, Zoo

```
public class Zoo{
   // This constructor has three parameters, name, city and
   state.
   public Zoo(String name, String city, String state) {
      System.out.println("Passed in Name is :" + name );
   }
   public static void main(String[] args) {
   // Following statement would create an object myZoo
   Zoo myZoo = new Zoo( “Woodland Park”, “Seattle”, “WA” );
   }
}
```

Accessing Instance Variables and Methods

- In Java, instance variables and methods are accessed via created objects:

```
/* First create an object */
objectReference = new ObjectReference();

/* Now access a variable as follows */
objectReference.variableName;

/* Now call a method as follows */
objectReference.methodName();
```

Accessing Instance Variables and Methods

```
public class Zoo{
     float zooSize;

     public Zoo(String name) {
          // This constructor has one parameter, name.
          System.out.println(”Zoo’s name is :" + name );
     }
     public void setSize(float size) {
          zooSize = size;
     }

     public float getSize( ) {
          return zooSize;
     }

     public static void main(String[] args) {
          Zoo myZoo = new Zoo( ”Woodland Park" );

          /* Call class method to set zoo size*/
          myZoo.setSize(55);

          /* Call another class method to get zoo size */
          float size = myZoo.getSize();
     }
}
```

Whirlwind Tour of Java

DATA TYPES IN JAVA

Data Types in Java

- Data type - a category of data values
  - Constrains the operations that can be performed

- Java distinguishes between
  - Primitive data types – store the actual values
  - Reference data types – store addresses to objects that they refer to

Primitive Data Types in Java

- Eight primitive data types are supported in Java:
  - byte – 8-bit signed two's complement integer (min. value -128, max value 127)
  - short – 16-bit signed two's complement integer (min. value -32,768, max. value 32,767)
  - int – 32-bit signed two's complement integer (min. value -2^{31}, max. value $2^{31} -1$)
  - long – 64-bit two's complement integer (min. value -2^{63}, max. value $2^{63}-1$)
  - float – single-precision 32-bit IEEE 754 floating point
  - double – double-precision 64-bit IEEE 754 floating point
  - boolean – only two possible values, true and false
  - char – single 16-bit Unicode character (min. value '\u0000' (0), max. value '\uffff' (65,535))

Reference Data Types in Java

In Java, reference data types are:

- Created using classes' constructors
- Used to access object of a declared type, or any object of a ***compatible*** type

- Some examples of reference data types:
  - `String`
  - `Scanner`
  - `Random`

Data Types in Java - Summary

- Question: what happens if we declare a variable, but don't initialize it?
- Answer: Most likely, the Java compiler will set those variables to reasonable default values

Data Type	Default Value
`byte`	0
`short`	0
`int`	0
`long`	0L
`float`	0.0f
`double`	0.0d
`char`	'\u0000'
`String (or any object)`	null
`boolean`	false

But don't do that → not initializing your data generally considered a bad programming style

Whirlwind Tour of Java

MODIFIER TYPES IN JAVA

Modifiers in Java

- Modifiers – keywords preceding the rest of the statement, used to change the meaning of the definitions of a class, method, or a variable
- Modifiers in Java can be:
  - Access control modifiers
  - Non-access control modifiers

Access-Control Modifiers in Java

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

Non-Access Control Modifiers in Java

- Non-access control modifiers in Java include:
  - `static` - for creating class methods and variables
  - `final` – for finalizing the implementations of classes, methods and variables
  - `abstract` – for creating abstract classes and methods
  - `synchronized` and `volatile` – used for threads

STRINGS IN JAVA

Strings in Java

- String - an object storing a sequence of text characters
  - *Unlike most other objects, a String object is not created with new**
    - String name = "text";
    - String name = expression;
- Example

```
int x = 5;
int y = 25;
String point = "(" + x + ", " + y +
")";
```

* This is not the whole story. `String` can be created as an object, with a keyword **`new`**. To learn more about why `Strings` are special in Java, please refer to Java Programming Tutorial, Java String is Special, [Online], https://www.ntu.edu.sg/home/ehchua/programming/java/J3d_String.html

Strings Indexes

- Characters of a string are numbered with 0-based indexes:
- Example:

```
    String name = ”NEU KHOURY";
```

Index	0	1	2	3	4	5	6	7	8	9
Char	N	E	U		K	H	O	U	R	Y

- The first character's index is always 0
- The last character's index is 1 less than the string's length
- The individual characters are values of type char

Strings Methods

- Example:
  - `String city = "Seattle";`
  - `System.out.println(city.length());   //7`

Method name	Description
`indexOf(str)`	Index where the start of the given string appears in this string (-1 if it is not there)
`length()`	Number of characters in the string
`substring(index1, index2),` `substring(index1)`	The characters in this string from *index1* (inclusive) to *index2* (exclusive). If *index2* omitted, grab till the end of string.
`toLowerCase()`	A new string with all lowercase letters
`toUpperCase()`	A new string with all uppercase letters

[Table credit: Paerson Education]

Modifying Strings

- Some methods (e.g., substring, toLowerCase, etc) create/return a new string, rather than modifying the current string

```
String s = ″northeastern";
s.toUpperCase();
System.out.println(s); // northeastern
```

- To modify a String variable, you must reassign it:

```
String s = ″northeastern";
s = s.toUpperCase();
System.out.println(s); // NORTHEASTERN
```

Comparing Strings

- Relational operators such as < and == fail on objects

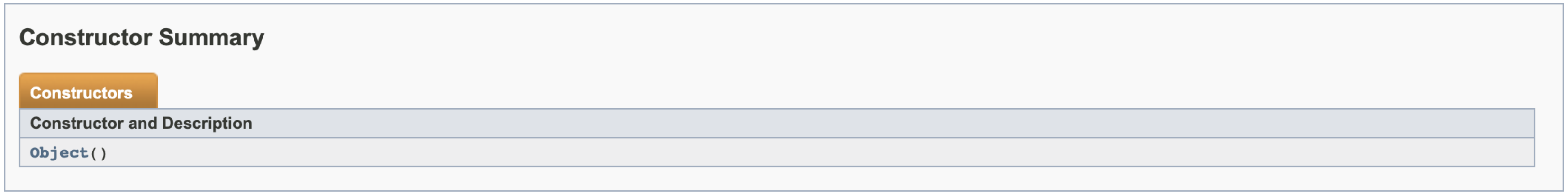
```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();

if (name == "Barney") {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

- This code will compile, but it will not print the song. Why?
    - == compares objects by references, so it often

gives **false** even when two Strings have the same letters

EQUALITY IN JAVA

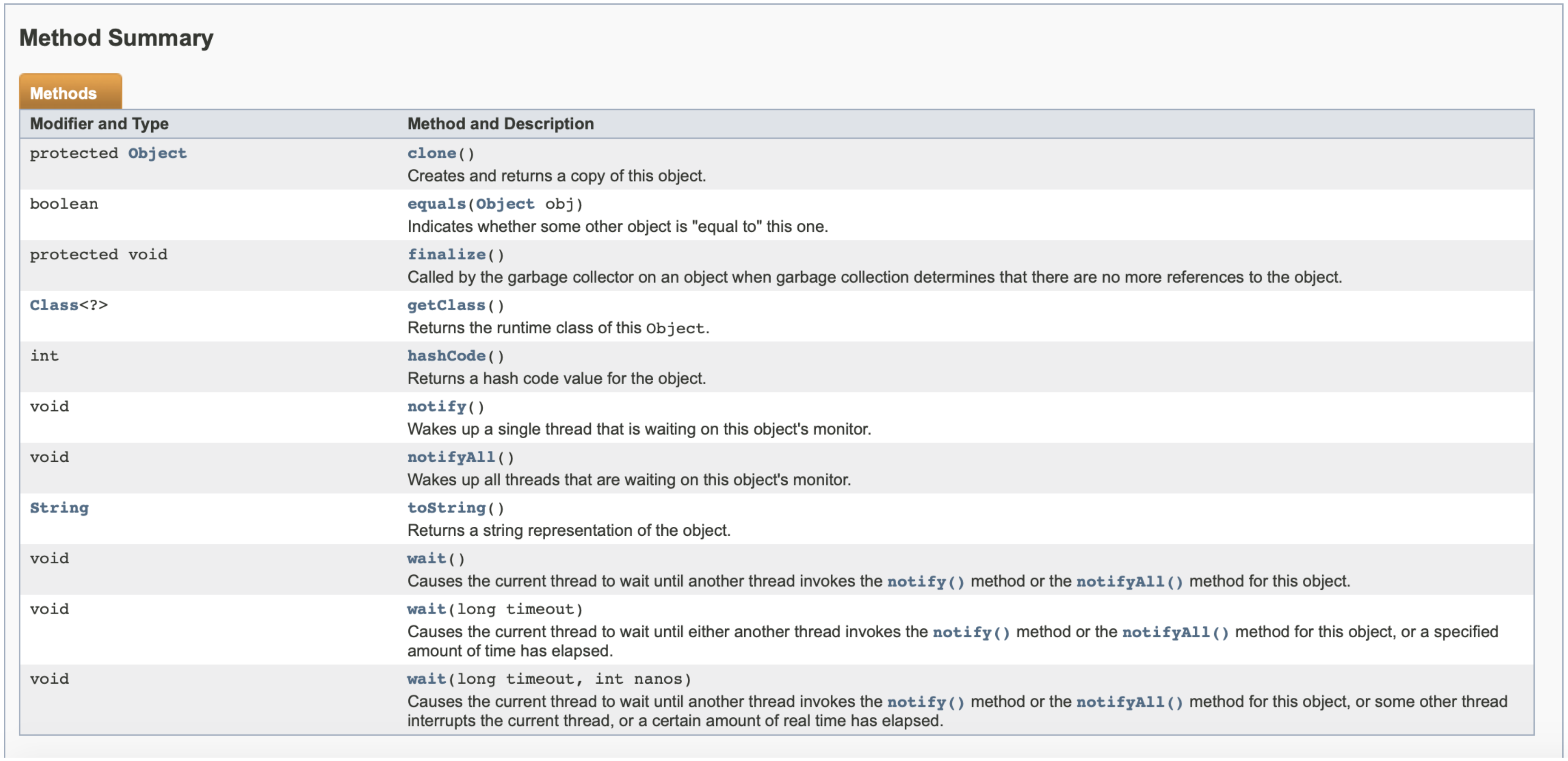
Class Object – Root of the Class Hierarchy



Constructor Summary

Constructors

Constructor and Description
`Object()`



Method Summary

Methods

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

TESTING CODE
UNIT TESTING

Software Reliability, Bugs, Testing, and Debugging

- Software reliability - probability that a software system will not cause failure under specified conditions
  - Measured by uptime, MTTF (mean time till failure), crash data.
- Bad news - bugs are inevitable in any complex software system
  - Industry estimates - 10-50 bugs per 1000 lines of code
  - A bug can be visible or can hide in your code until much later
- Testing - a systematic attempt to reveal errors
  - Failed test: an error was demonstrated
  - Passed test: no error was found (for this particular situation)

Difficulties with Testing

- Perception by some developers and managers:
  - Testing is seen as a novice's job
  - Assigned to the least experienced team members
  - Done as an afterthought (if at all)
  - "My code is good; it won't have bugs. I don't need to test it.”
  - "I'll just find the bugs by running the client program."

- Limitations of what testing can show you:
  - It is impossible to completely test a system
  - Testing does not always directly reveal the actual bugs in the code
  - Testing does not prove the absence of errors in software

Testing Your Code

- Writing tests - an essential part of code design and implementation
  - The most important skill in writing tests - determining what to test, and how to test

- Idea: if there is a bug in code, you want to catch it at compile time
  - That means that that client code that incorrectly uses your code should ideally produce compile-time errors

- Ideal workflow: write interface → write an empty implementation → write test cases

Kinds of Tests

- Unit tests - code that tests the smallest components of a program - individual functions, classes, or interfaces, to confirm that they work as expected
  - Used to confirm that algorithms seem to work as expected on their inputs, and that edge cases are properly handled
- Regression tests – test written as soon as a bug is noticed and fixed, to ensure that the bug can never creep back into the program inadvertently
- Integration tests – code that tests larger units of functionality, or libraries (trickier to write)
- Randomized or “fuzz” tests - designed to rapidly explore a wider space of potential inputs than can easily be written manually
  - Typically used to check the **robustness of a program’s error handling**, to see whether it holds up without **crashing even under truly odd inputs**

White Box Testing

- White-box testing (clear box, glass box, transparent box, and structural testing)
  - Internal perspective of the software under test
  - Tester looks inside the software that is being tested, and uses that implementation knowledge as a part of the testing process
  - Example: if an exception is thrown under certain conditions, then a white box test might want to reproduce those conditions

- Some advantages of white-box testing:
  - Efficient in finding errors and problems
  - Allows finding hidden errors
  - Helps optimizing the code
  - Due to required internal knowledge of the software, maximum coverage is obtained

- Some disadvantages of white-box testing:
  - Might not find unimplemented or missing features
  - Requires high level knowledge of internals of the software under test
  - Requires code access
- White-box testing is almost always automated, and in most cases has the form of unit tests

Black Box Testing

- Black-box testing (functional testing)
  - External perspective of the software under test
  - Tester treats software under test as a black-box, without knowing its internals
  - Tests are executed using software interfaces, and trying to ensure that they work as expected (idea: as long as functionality of interfaces remains unchanged, tests should pass even if internals are changed)
- Some advantages of black-box testing:
  - Efficient for large segments of code
  - Code access is not required
  - Separation between user’s and developer’s perspectives
- Some disadvantages of black-box testing:
  - Limited coverage since only a fraction of test scenarios is performed
  - Inefficient testing due to tester’s lack of knowledge about software internals
  - Blind coverage since tester has limited knowledge about the application

Unit Testing

- Unit testing - search for errors in a subsystem in isolation
  - A "subsystem" typically means a particular 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 particular 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

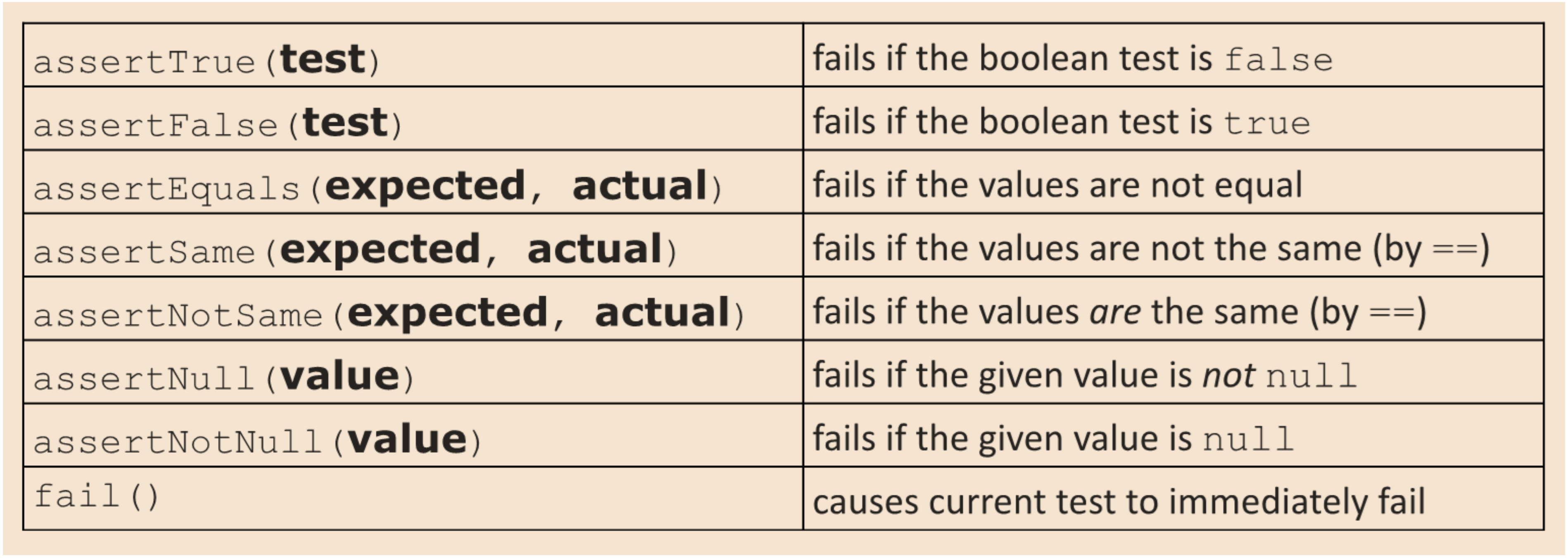
- Methods to run before/after each test case method is called:

```
@Before
public void setUpTestCase() { ... }
@After
public void tearDownTestCase() { ... }
```

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

```
@BeforeClass
public static void setUp() { ... }
@AfterClass
public static void tearDown() { ... }
```

JUnit Assertion Methods



`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expected**`, `**actual**`)`	fails if the values are not equal
`assertSame(`**expected**`, `**actual**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**expected**`, `**actual**`)`	fails if the values *are* the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. assertEquals("message", expected, actual)

JUnit – Tips for Testing

- You cannot test every possible input, parameter value…
  - So you must think of a limited set of tests likely to expose bugs
- Think about boundary cases:
  - Positive; zero; negative numbers
  - Right at the edge of an array or collection's size
- Think about empty cases and error cases:
  - 0, -1, null; an empty list or array
- Test behavior in combination
  - Maybe add usually works, but fails after you call remove
  - Make multiple calls; maybe size fails the second time only

JUnit – Tips for Testing

- Test one thing at a time per test method
  - 10 small tests are much better than 1 test 10x as large
- Each test method should have few (likely 1) assert statements
  - If you assert many things, the first that fails stops the test
  - You won't know whether a later assertion would have failed
- Tests should avoid logic
  - Minimize if/else,loops,switch,etc
  - Avoid try/catch
  - If it's supposed to throw, use **assertThrows** ... if not, let JUnit catch it
- Torture tests are okay, but only in addition to simple tests

JUnit – Things to Avoid

- "Smells" (bad things to avoid) in tests:
  - **Constrained test order:** Test *A* must run before Test *B*
    - (usually a misguided attempt to test order/flow)
  - **Tests call each other:** Test *A* calls Test *B*'s method
    - (calling a shared helper is OK, though)
  - **Mutable shared state:** Tests *A*/*B* both use a shared object.
    - (If *A* breaks it, what happens to *B*?)

JUnit – Summary

- Tests need failure atomicity (ability to know exactly what failed)
- Each test should have a clear, long, descriptive name
    - Assertions should always have clear messages to know what failed
    - Write many small tests, not one big test
        - Each test should have roughly just 1 assertion at its end
- Test for expected errors / exception
- Choose a descriptive assert method, not always `assertTrue`
- Choose representative test cases from equivalent input classes
- Avoid complex logic in test methods if possible
- Use helpers, @Before to reduce redundancy between tests

DEALING WITH UNEXPECTED SITUATIONS. EXCEPTIONS

Exceptions

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

Types of Exceptions

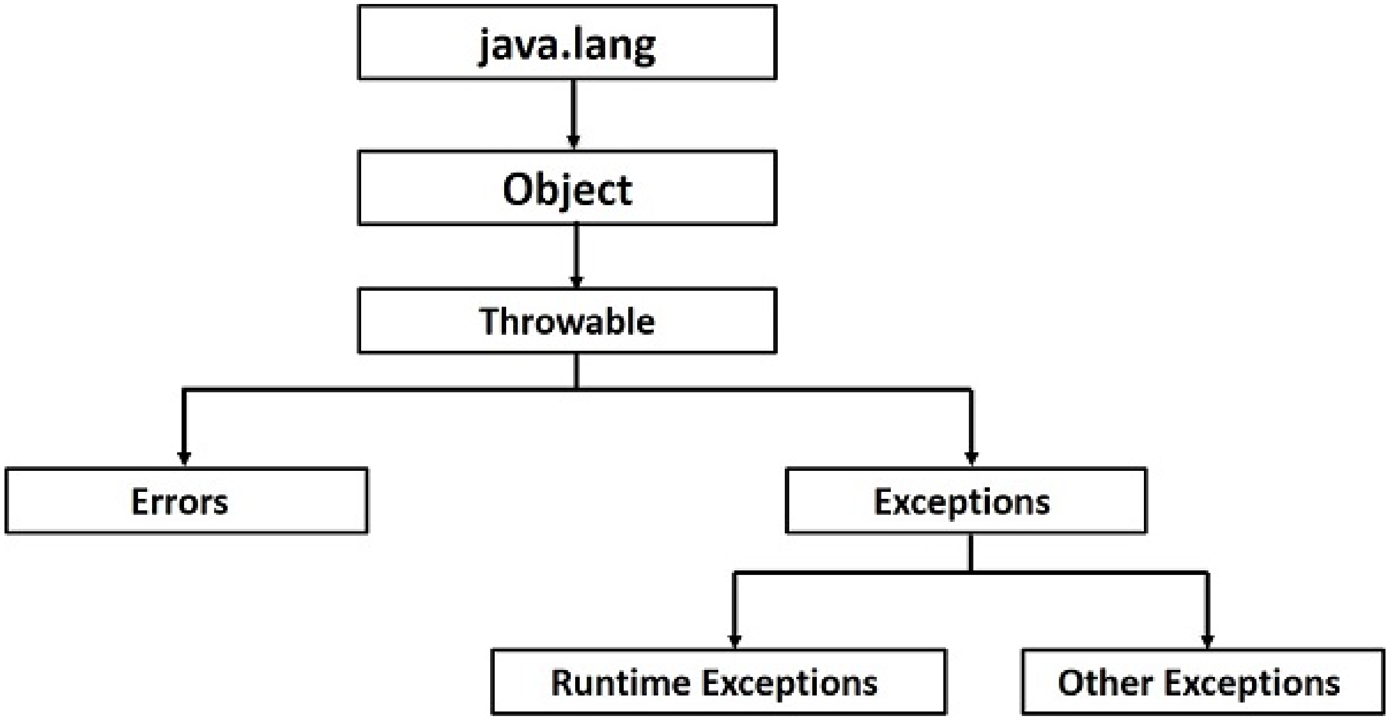
- An exception occurs when something unexpected happens during the program execution

- 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]

Using Exceptions – A General Recipe

- A general procedure whenever we write a new method:
  - Design/write a method signature
  - Write the method body so that it works correctly under ideal circumstances
  - Carefully think about all non-ideal situations that may occur, other than ideal circumstances
  - For each 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)
    - Choose an appropriate exception type for the error:
      - Declare that this method throws this exception
      - Throw this exception appropriately

Catching Exceptions

- 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 Exceptions – 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.
}
```

Writing a Method with Exception

```java
/**
   * 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;
  }
```

Calling a Method with Exception

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

Testing a Method with Exception

```
@Test
public void testIllegalDiscount() {
  float discountedPrice;

  try {
    discountedPrice = beaches.salePrice(20);
    assertEquals(16.0f,discountedPrice,0.01);
  }
  catch (IllegalArgumentException e) {
    fail("An exception should not have been thrown");
  }

  try {
    discountedPrice = beaches.salePrice(-20);
    fail("An exception should have been thrown");
  }
  catch (IllegalArgumentException e) {

  }
}
```

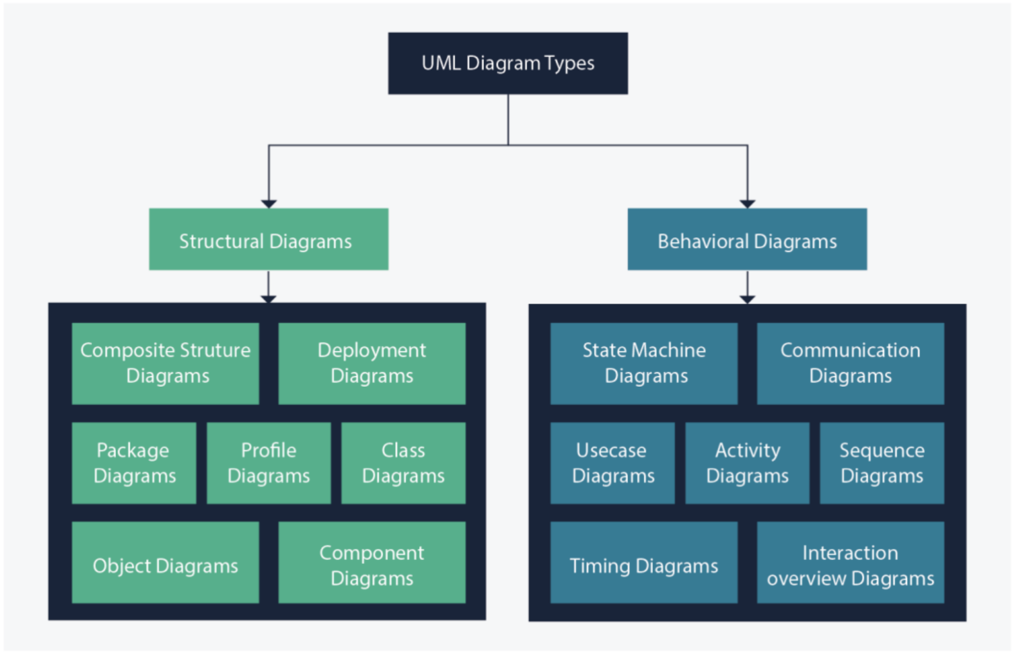
Writing Your Own Exceptions

- 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

UML DIAGRAMS

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 Diagrams

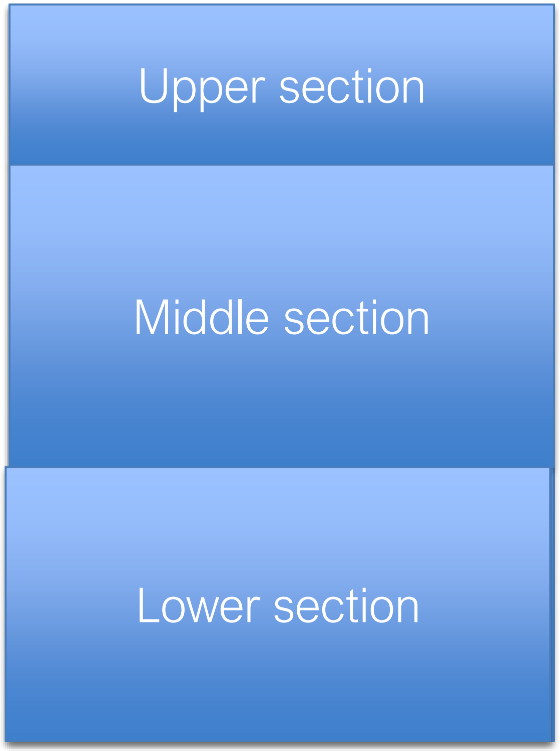
- Class UML diagram – the main building block of any object oriented solution, showing:
  - The classes involved in the solution
  - Relationships between those classes
  - Fields (states/attributes) of every class
  - Methods (operations/behavior) of every class

- Some benefits of Class UML diagrams:

1. Allow for a better understanding of a problem through a graphical representation
2. Provide an implementation-independent description of data types (objects) that are used in a system, and are later passed between its components
3. Provide a detailed representation of 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

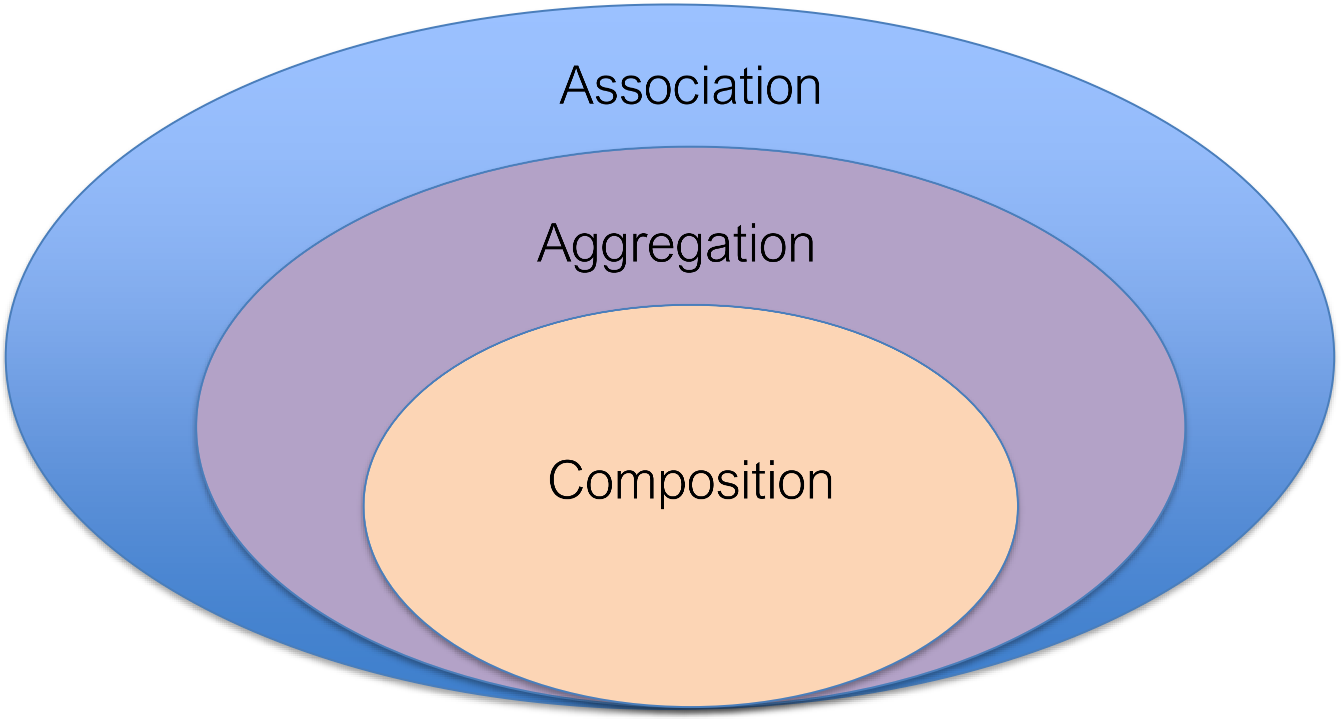


Class UML Diagram - Relationships

- 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 Books
    - 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

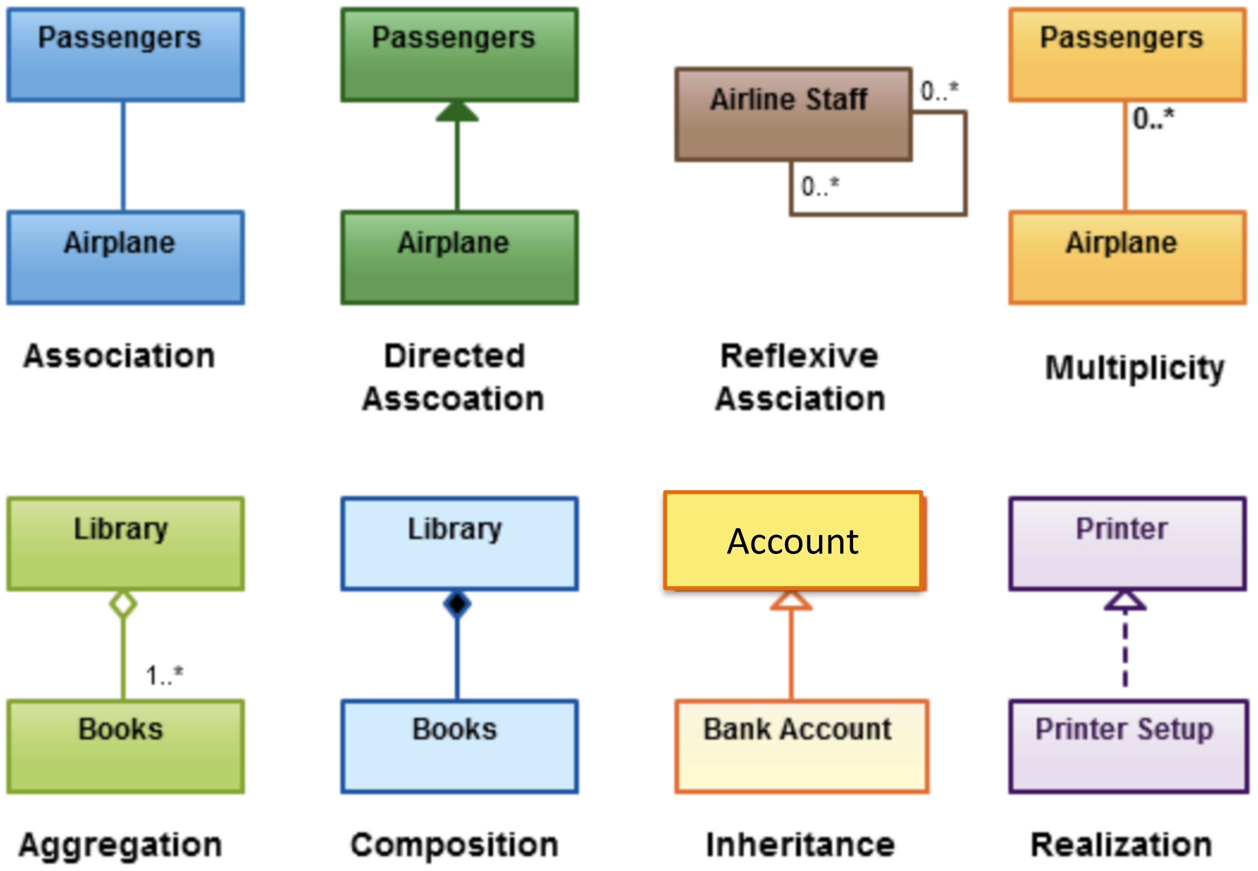
Class UML Diagram - Relationships

- Association, Aggregations and Composition



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

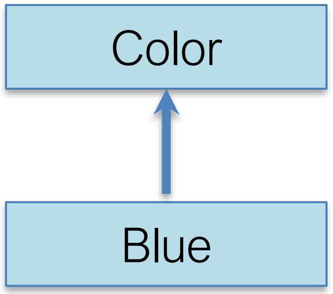
Class UML Diagram - Relationships



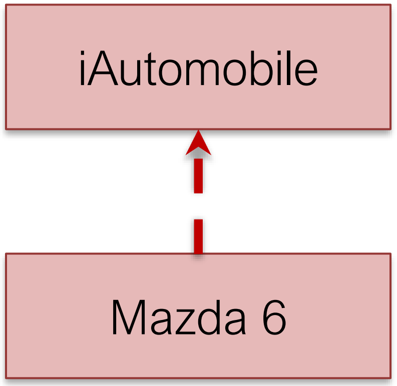
[Picture credit: https://creately.com/blog/diagrams/class-diagram-relationships/]

Class UML Diagram - Relationships

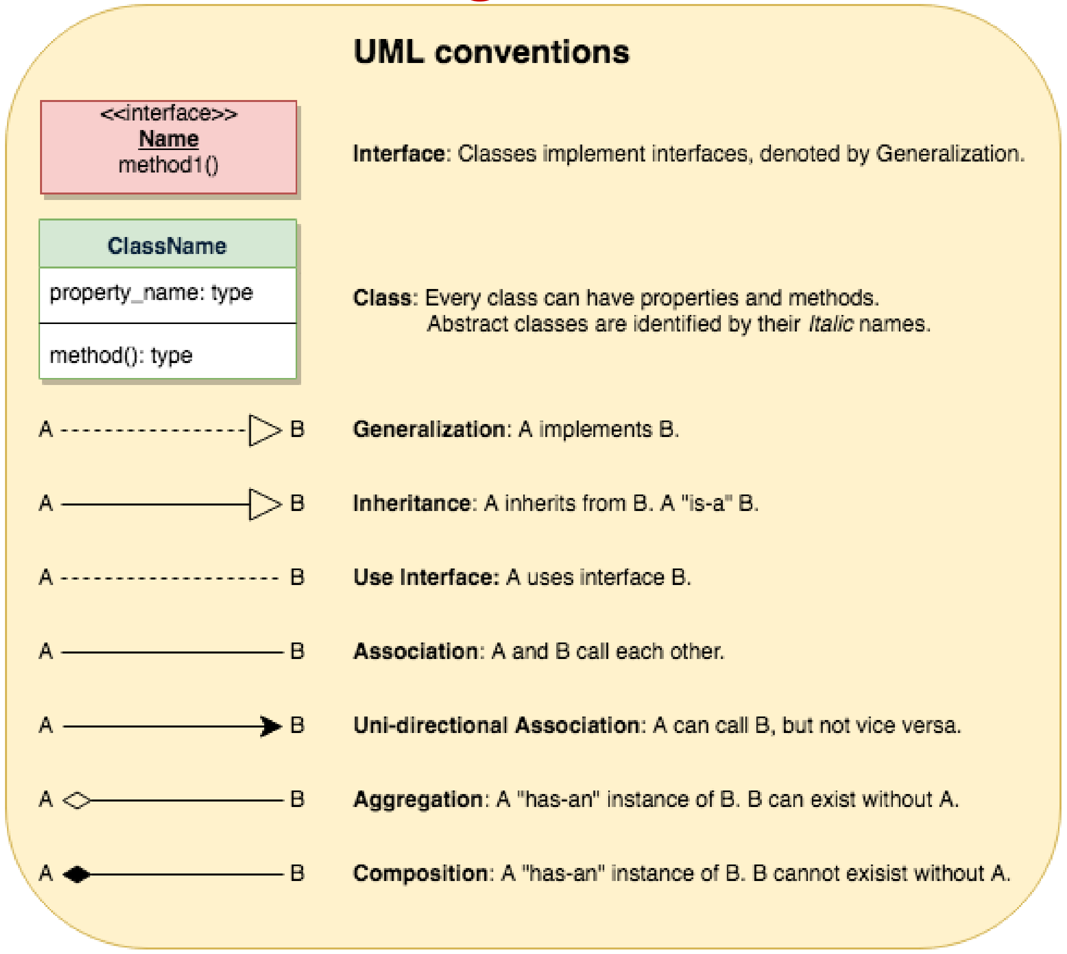
- Inheritance/generalization - a type of relationship wherein one associated class is a child of another, by virtue of assuming the same functionalities of the parent class (the child class is a specific type of the parent class)



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



Class UML Diagram - Summary



UML conventions

<<interface>> **Name** method1()	**Interface**: Classes implement interfaces, denoted by Generalization.
ClassName / property_name: type / method(): type	**Class**: Every class can have properties and methods. Abstract classes are identified by their *Italic* names.
A - - - -▷ B	**Generalization**: A implements B.
A ——▷ B	**Inheritance**: A inherits from B. A "is-a" B.
A - - - - B	**Use Interface:** A uses interface B.
A —— B	**Association**: A and B call each other.
A ——➤ B	**Uni-directional Association**: A can call B, but not vice versa.
A ◇—— B	**Aggregation**: A "has-an" instance of B. B can exist without A.
A ◆—— B	**Composition**: A "has-an" instance of B. B cannot exisist without A.

[Figure credit: www.education.io]

COMMENTING AND DOCUMENTATION JAVADOC

Commenting and Documentation

- Documentation - comments explaining your design and purpose
- General documentation rules:
- Classes - explain in a sentence or two what a class represents
  - Semantic details - what the class represents from the problem statement point of view
  - Technical details - things that you would like to know before, or in order to interact with the class
  - 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
  - List all arguments and what they represent
  - Explain what the method returns
  - 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 Comment

- 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)
    @exception (@throws is a synonym added in Javadoc 1.2)
    @see
    @since
    @serial (or @serialField or @serialData)
    @deprecated (see How and When To Deprecate APIs)

Structure of a Javadoc Comment

- 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

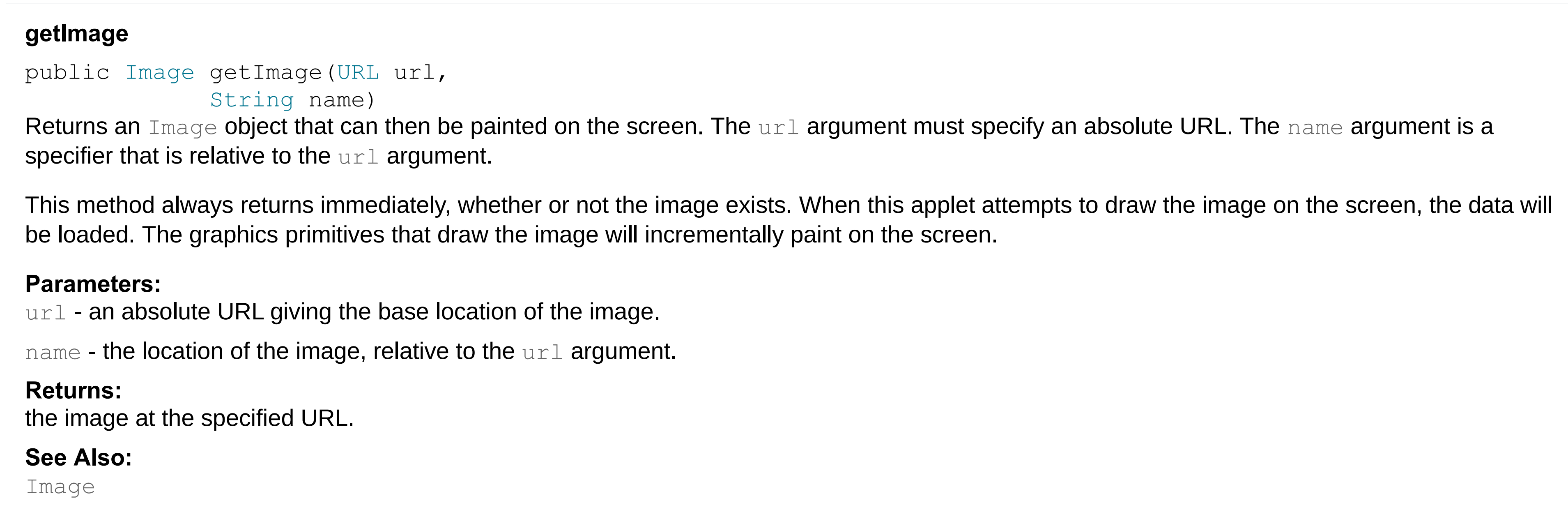
Javadoc Comment - Example

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
```

[Example credit: http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html]

Javadoc Comment - Example

Resulting Javadoc:



getImage

```
public Image getImage(URL url,
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:
`url` - an absolute URL giving the base location of the image.

`name` - the location of the image, relative to the `url` argument.

Returns:
the image at the specified URL.

See Also:
`Image`

[Example credit: http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html]

Javadoc Comment - Example

- Some notes about the example:
  - Each line indented to align with the code below the comment
  - The first line contains the begin-comment delimiter (/**)
  - The first sentence - a short summary of the method, that Javadoc automatically places it in the method summary table (and index)
  - Inline tag {@link URL} - converts to an HTML hyperlink pointing to the documentation for the URL class, and it can be used anywhere that a comment can be written
  - Paragraph separator <p> - if you have more than one paragraph in the doc comment, you can separate them with the paragraphs with a <p>
  - Convention - insert a blank comment line between the description and the list of tags
  - End of description - the first line that begins with an "@" character ends the description. There is only one description block per doc comment; you cannot continue the description following block tags.
  - The last line contains the end-comment delimiter (*/) (note that unlike the begin-comment delimiter, the end-comment contains only a single asterisk)

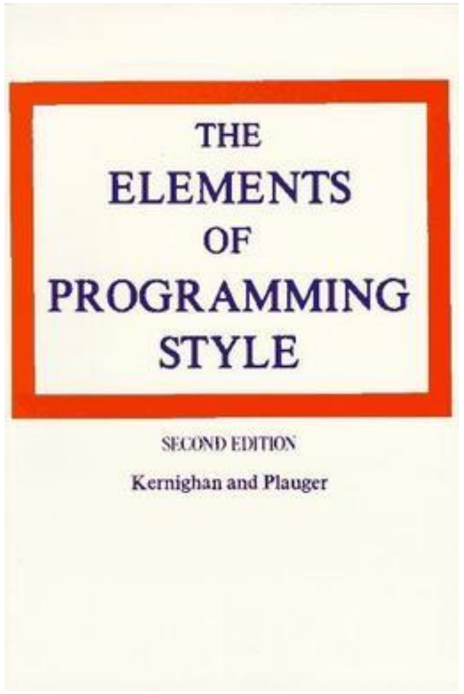
- For more examples, see Simple Examples

CODESTYLE (PROGRAMMING STYLE)

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's with the Elements of Programming Style


THE
ELEMENTS
OF
PROGRAMMING
STYLE

SECOND EDITION

Kernighan and Plauger

[Picuture 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
  - Goal: Improve readability

Hello Java 17 – IntelliJ - Tools

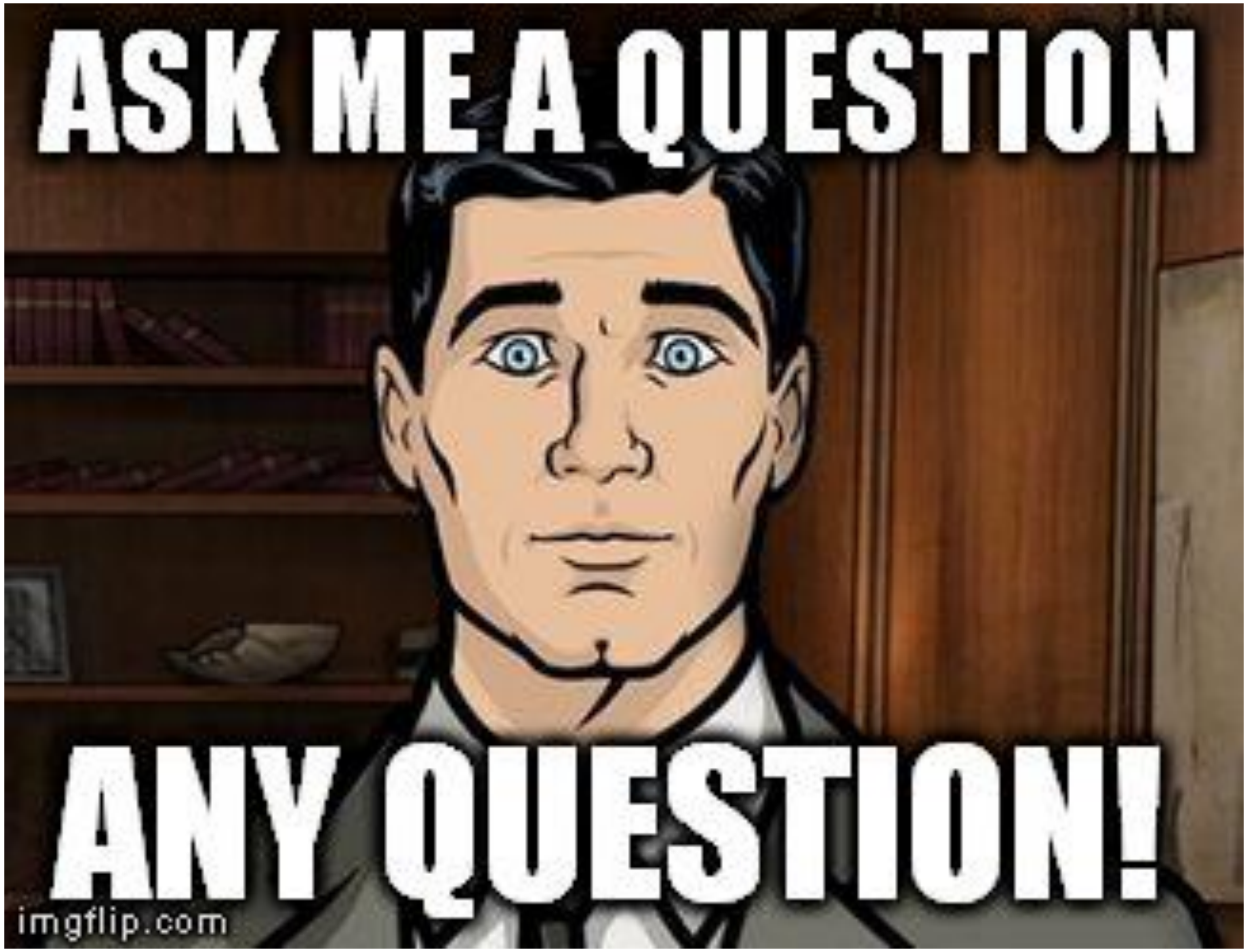
EXPLORE

RANDOM BITS

Random Things

- LTS (Long Term Support): Java moved to guaranteeing long term support on specified version. Previous is Java 11. Java 17 is latest.
- SDK
- Garbage Collection
- Generics
- Lambda Functions
- Pass by reference
- JVM
- …

Your Questions



[Meme credit: imgflip.com]

References and Reading Material

- How to Design Classes (HtDC), Chapters 1-3
- 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)
- 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 – Exceptions (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)