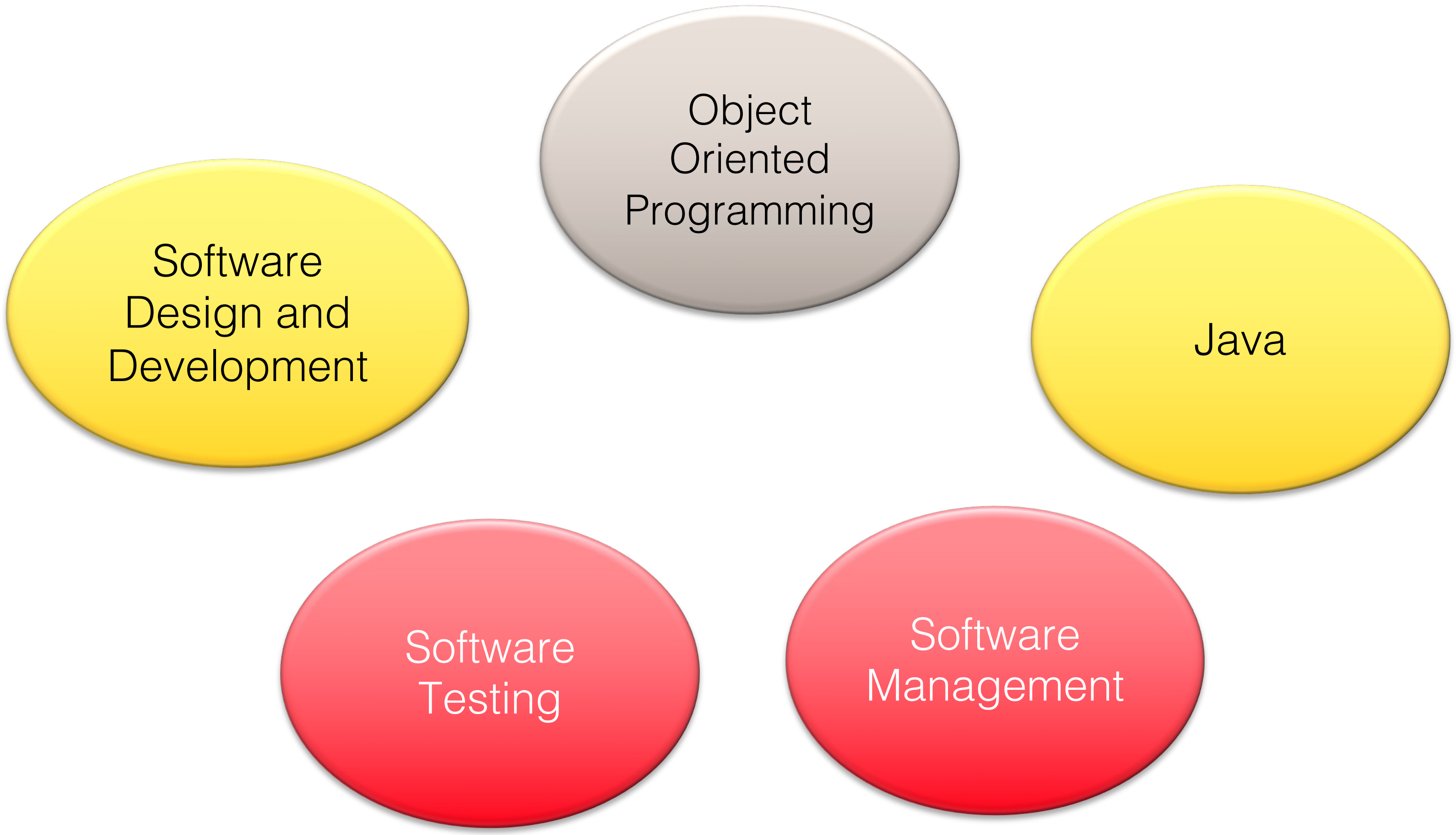
CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2024

LECTURE 1

Tamara Bonaci
t.bonaci@northeastern.edu

ONE MORE TIME: WHAT IS CS 5004, SPRING 2024?



HIGH QUALITY SOFTWARE

CS 5004, SPRING 2024 – LECTURE 1

HIGH QUALITY SOFTWARE

- High quality software should be:
  - Correct
  - Comprehensible
  - Modifiable

HIGH QUALITY SOFTWARE: CORRECT

- Meet functional requirements
  - Pass test cases
- But programming is not math…
  - No one answer
    - But there are good ones and bad ones ☺
  - No single design method or approach
- Programming is a design exercise…
  - Apply design principles
  - Apply best practices (such as design patterns)
  - Justify and explain your thinking

HIGH QUALITY SOFTWARE: COMPREHENSIBLE

- Code has two equally important audiences:
  - CPU and systems
  - Other engineers
- Code should be:
  - Easy for others to understand
  - Well documented
- This will be tested in codewalks
  - You'll need to explain your design and code to TAs and Professors

HIGH QUALITY SOFTWARE: MODIFIABLE

- Software systems always change and evolve
  - Your code should be comprehensible, so other engineers can use and modify it

- Design principles make it possible to build modifiable software
  - But there are always trade-offs
  - Some changes are easier to make than others
    - And some will be hard/impossible
  - The art of design is to anticipate likely/most common changes and accommodate those

SOFTWARE ENGINEERING AND PRACTICE

- Good software is not just the right output
  - Many other goals exist
- "Software engineering" promotes the creation of good software, in all its aspects
  - Directly code-related: class and method design
  - External: documentation, style
  - Higher-level: e.g., system architecture
- Software quality is important in this class and in the profession (but it doesn't happen over night)

PROGRAMMING PARADIGMS

CS 5004, SPRING 2024 – LECTURE 1

PROGRAMMING PARADIGMS

- Programming paradigm - general approach used to implement a program
  - A programming style that indicates the approach on how a solution is implemented in a programming language
- Programming languages are typically design with at least one paradigm in mind
  - But a language can typically accommodate more than one programming style
  - It is also possible to use one programming style to encode another

POPULAR PROGRAMMING PARADIGMS

- Imperative (procedural) paradigm
- Applicative (functional) paradigm
- Object oriented paradigm
- Logical programming paradigm
- Event-driven paradigm
- Actor-oriented paradigm
- Meta-programming

IMPERATIVE PROGRAMMING PARADIGM

- Imperative programs are made up of procedures and data
  - Procedures are made up of a sequence of statements
  - Each statement may alter the existing data in place (typically this means there is no return value from our procedure)

- Conceptually, we can then think of an imperative program's execution as a machine that executes each statement in the appropriate sequence, and a store that holds all the data that each statement will alter (or mutate)

- Some popular procedural languages: C, Pascal, and Ada

FUNCTIONAL PROGRAMMING PARADIGM

- Functional programs are made up of functions and data as values
- Functions are made up of one expression
- A function takes input values and returns output values

- Conceptually, a functional program's execution is an evaluation, much like the evaluation process in simple mathematical expressions

- Some popular functional languages: Lisp, Scheme, ML, Haskell

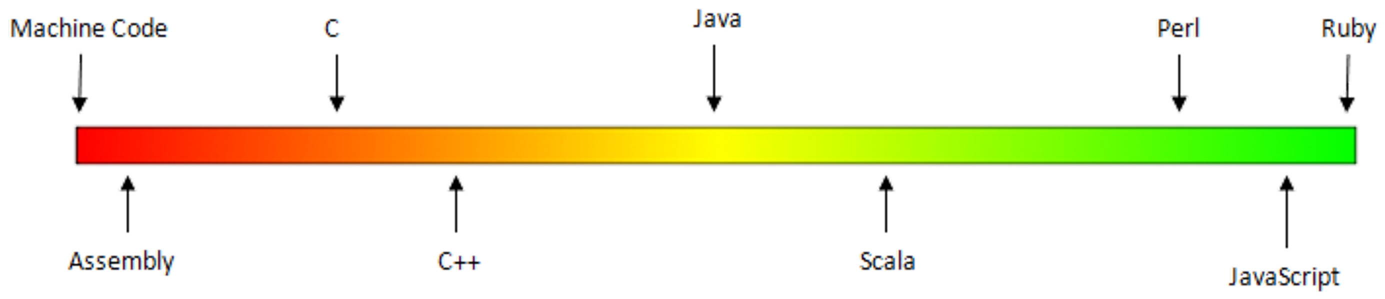
OBJECT-ORIENTED PROGRAMMING PARADIGM

- OO programs are made up of objects that communicate with each other by messages

- Class-based OO program - notion of a class that is made up of fields (data) and methods (messages that this class understands)
  - From a class, we can then create an object that has its own set of fields but shares the same methods
  - A method is made up of a sequence of statement and/or expressions that may manipulate the objects' fields and may return an object as a result of executing that method

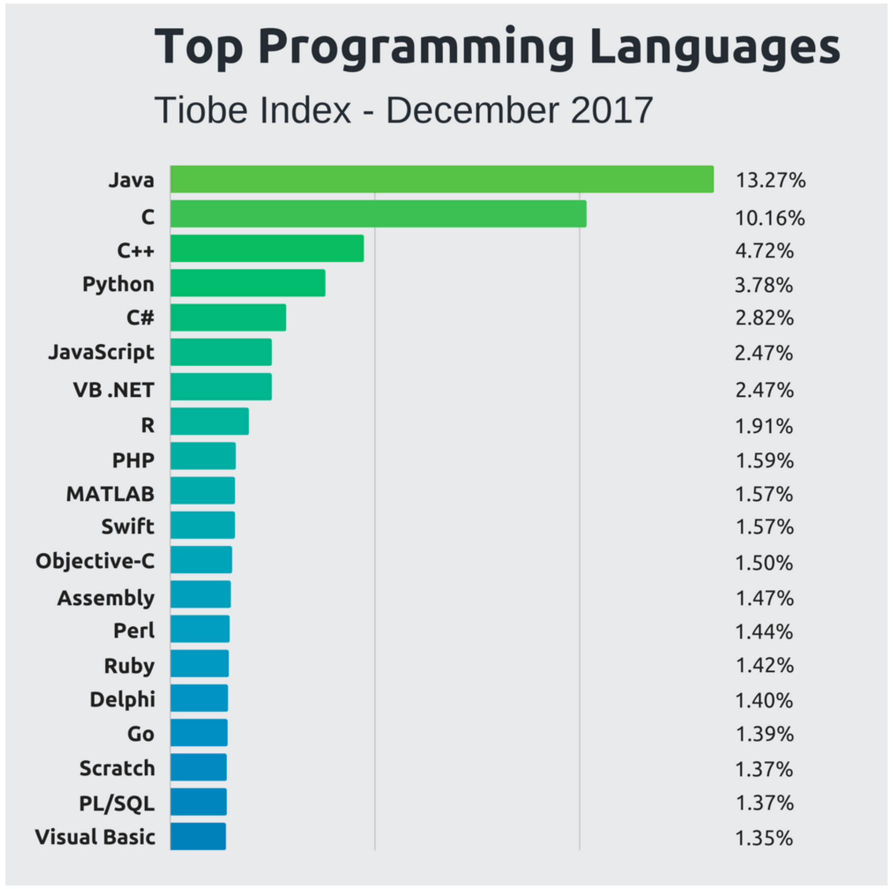
- Some popular OOD languages: Java, C++ , C#, Eiffel, Smalltalk

SOME MODERN LANGUAGES



- Procedural languages: programs are a series of command
    - 1970 - Pascal - designed for education
    - 1972 - C - low-level, operating systems and device drivers
- Object-oriented languages: programs interact using "objects"
    - 1985 - C++
    - 1995 - Java
        - Designed for embedded systems, web apps/servers
        - Runs on many platforms (Windows, Mac, Linux, cell phones...)

MOST POPULAR PROGRAMMING LANGUAGES



[Pictures credit: https://stackify.com/popular-programming-languages-2018/
]

DATA TYPES IN JAVA

CS 5004, SPRING 2024 – LECTURE 1

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 complelent 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))

OOD – THE BEGINNING

CS 5004, SPRING 2024 – LECTURE 1

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)

OBJECT-ORIENTED DESIGN: THE BEGINNING

- Identify objects
- Identify properties
- Identify responsibilities

- Code example – Classes `Person, Book and Zoo`

CLASSES AND CONSTRUCTORS IN JAVA

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

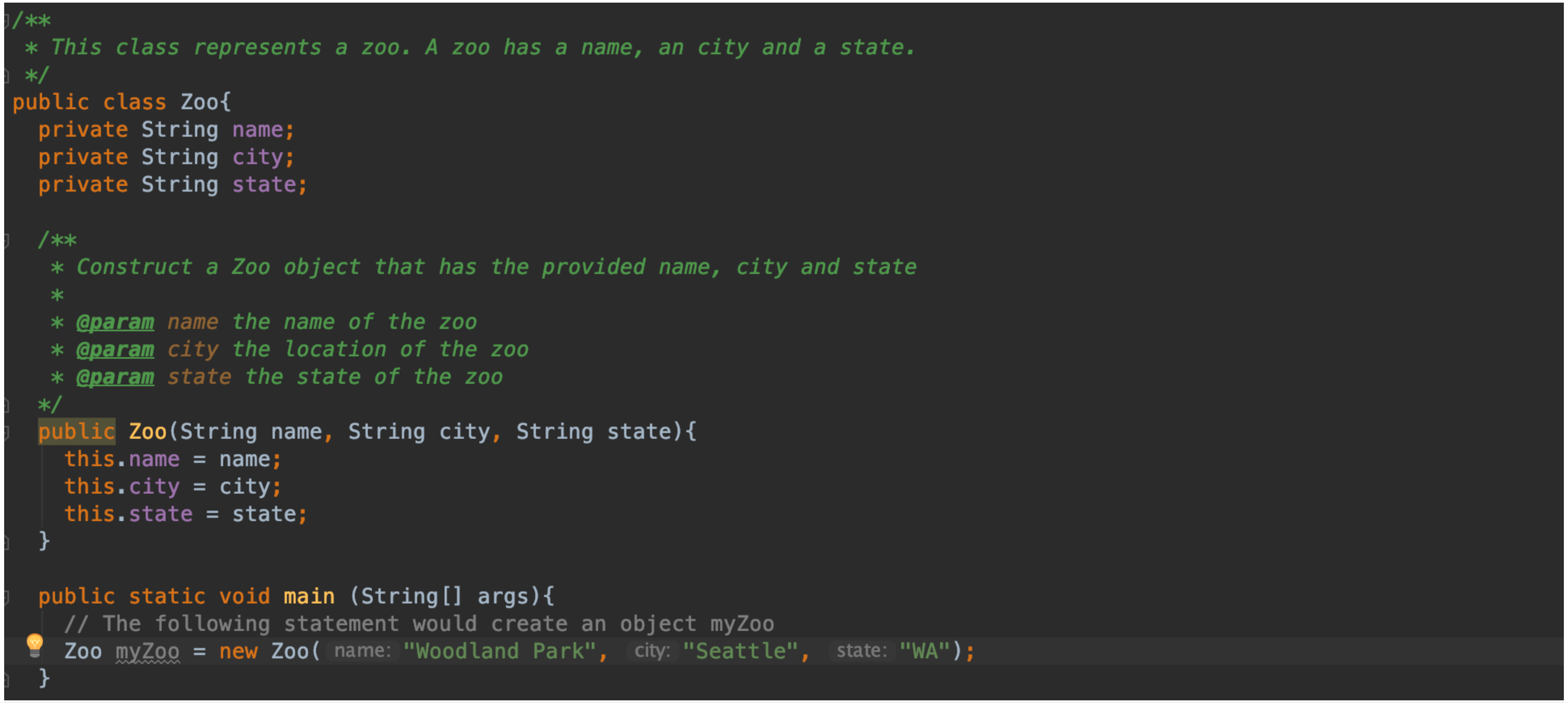
- Every class has a constructor
  - In Java, if we don't explicitly write a constructor, Java compiler builds a default constructor for that class
  - But it is a good practice to write a constructor, even an empty one

CREATING AN OBJECT IN JAVA

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

CREATING AN OBJECT IN JAVA

- Example: creating an object Zoo:



```
/**
 * This class represents a zoo. A zoo has a name, an city and a state.
 */
public class Zoo{
  private String name;
  private String city;
  private String state;

  /**
   * Construct a Zoo object that has the provided name, city and state
   *
   * @param name the name of the zoo
   * @param city the location of the zoo
   * @param state the state of the zoo
   */
  public Zoo(String name, String city, String state){
    this.name = name;
    this.city = city;
    this.state = state;
  }

  public static void main (String[] args){
    // The following statement would create an object myZoo
    Zoo myZoo = new Zoo( name: "Woodland Park",  city: "Seattle",  state: "WA");
  }
```

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:
  - 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`**)

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

TESTING CODE. UNIT TESTING

CS 5004, SPRING 2024 – LECTURE 1

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

TESTING YOUR CODE – SOME IDEAS

- Idea: if there is a bug in your 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

- Look at each method of the interface in isolation:
  - What behavior you expect when all inputs are correct and as expected?
  - What are all possible correct and incorrect parameters, and how do you check that the method behaves as expected in each situation?
  - How do you verify that an implementation of the method actually fulfills these objectives?
  - How do you reproduce exceptional cases so that you can test them?
  - What are the sequences in which various methods of the interface might be called?
  - Is there a “correct” sequence of calling them?
  - What happens when they are called “out of sequence,” and what should happen?

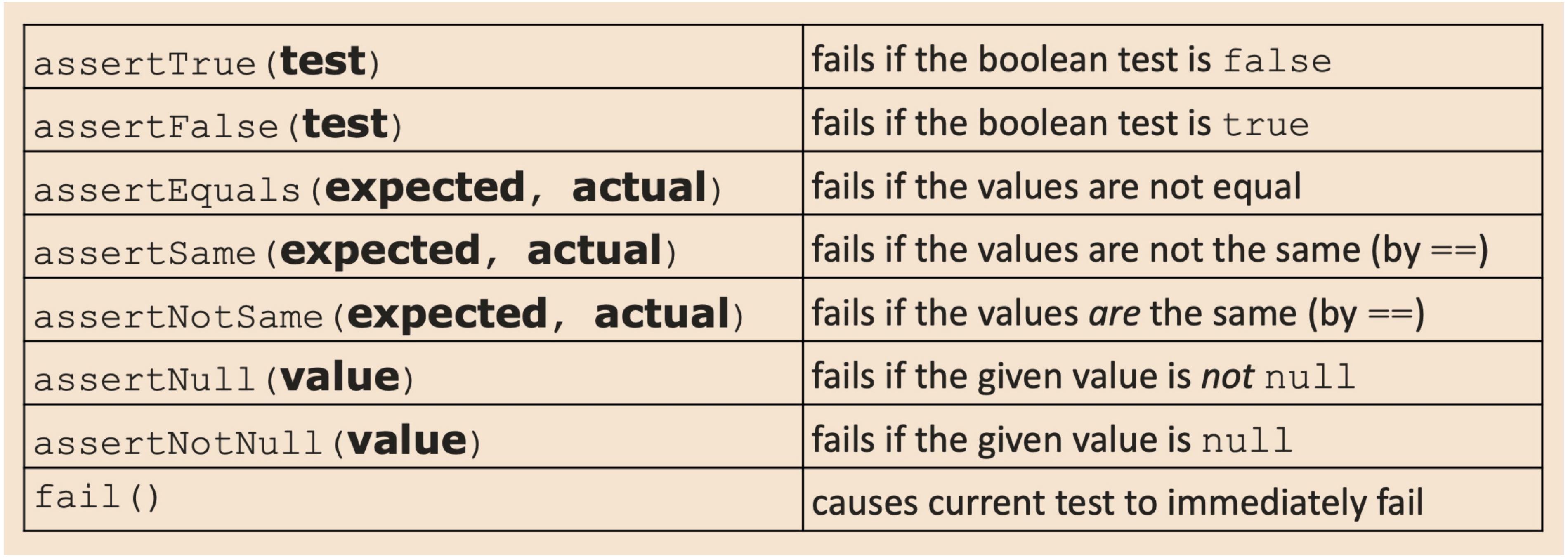
KINDS OF TESTS

- 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**

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 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 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 – 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 expected= ... 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

OBJECT-ORIENTED DESIGN AND ANALYSIS

CS 5004, SPRING 2024 – LECTURE 1

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]

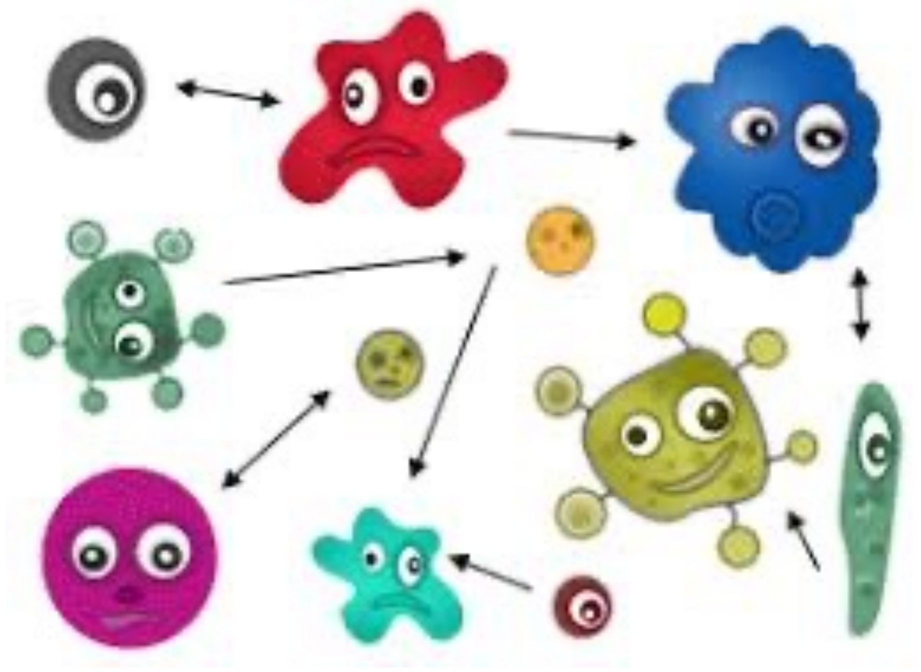
Encapsulation idea – data types and methods operating on that data coupled within an object/class

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

EXAMPLE

CS 5004, SPRING 2024 – LECTURE 1

OBJECT ORIENTED DESIGN - EXAMPLE

- Identify objects
- Identify properties
- Identify responsibilities

- Example: food delivery mobile app

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Objects:
  - Customers
  - Drivers
  - Restaurants
  - Menus
  - Order

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Customers - properties:
  - Name
  - Address
  - Phone number
  - E-mail address
  - Order

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Driver - properties:
  - Hours of operation
  - Current location
  - Next delivery

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Restaurant - properties:
  - Cuisine
  - Address
  - Hours of operation (open/close)
  - “Priciness” ($, $$, $$$)
  - Customer rating
  - Menu

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Menu - properties:
  - Offered meals
  - Meal prices

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Order - properties:
  - Total quantity
  - Total price
  - Paid/Not paid

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Customer - responsibilities:
  - Search for restaurants
  - Choose a restaurant
  - Check the menu
  - Select meals from the menu
  - Make an order
  - Pay for an order
  - Track an order
  - Cancel an order

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Driver - responsibilities:
  - Receive an order
  - Deliver an order
  - Change paid/not paid status of an order

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Restaurant - responsibilities:
  - Receive an order
  - Dispatch an order to a driver
  - Track the order

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Menu - responsibilities:
  - Update meals
  - Update meal prices

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

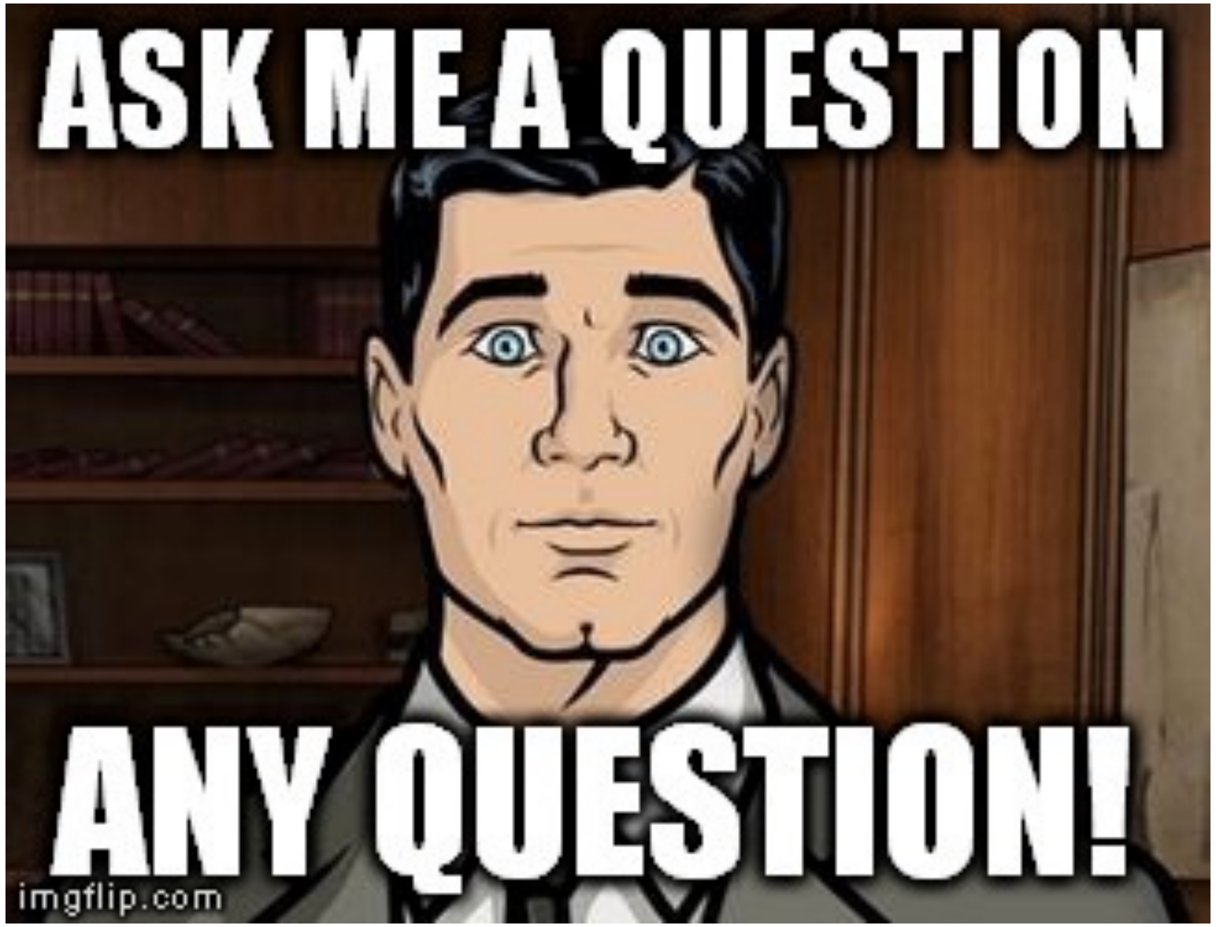
OBJECT ORIENTED DESIGN - EXAMPLE

- Example: food delivery mobile app

- Order - responsibilities:
  - Update quantity
  - Cancel order
  - Track order status

[Pictures credit: http://www.charlottemagazine.com/Charlotte-Magazine/October-2016/Rating-Charlottes-Food-Delivery-Services/]

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

[Meme credit: imgflip.com]