Module 1

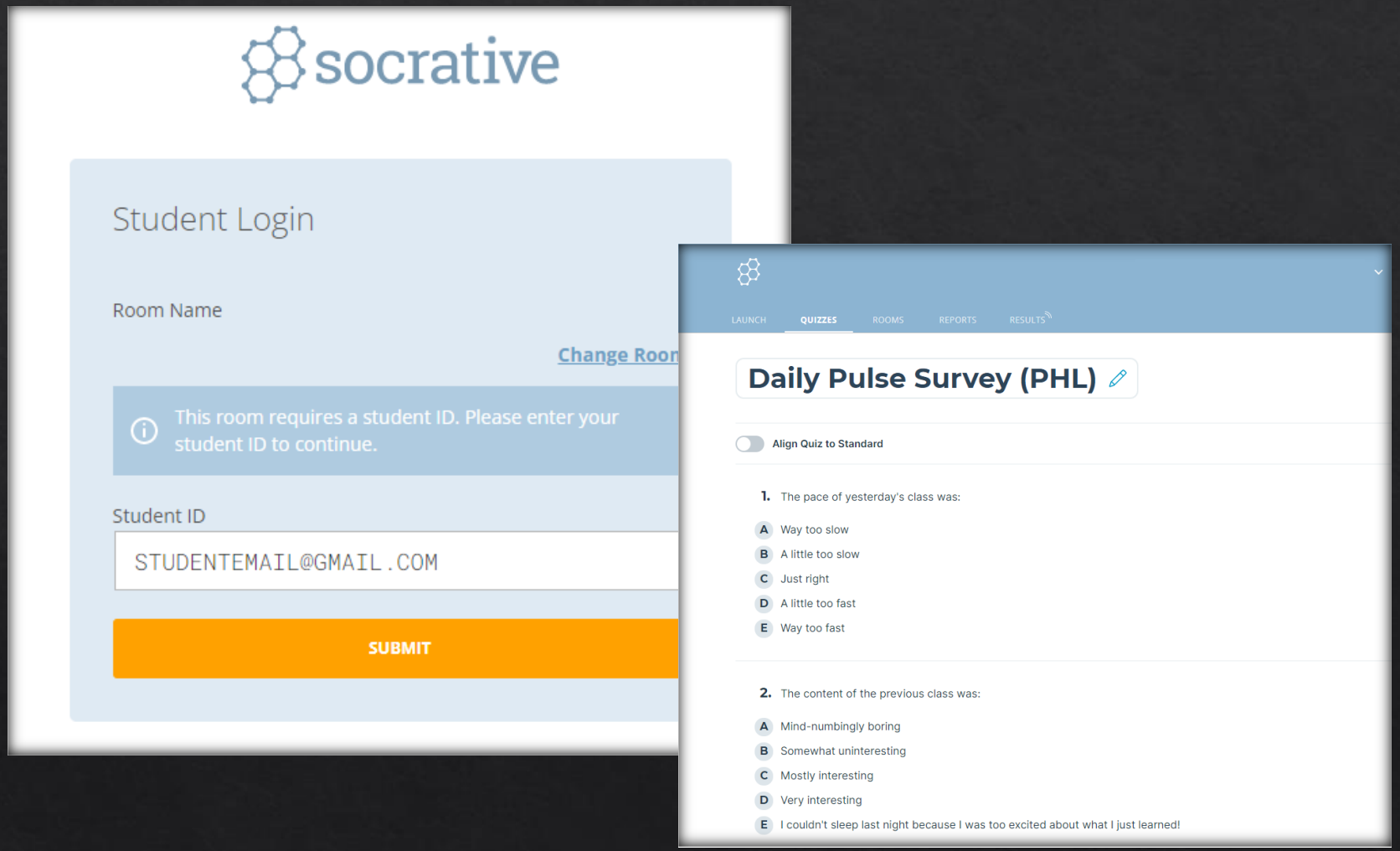
WEEK 3 REVIEW

JAVA – MODULE 1, DAY 15

TECH ELEVATOR

Announcements

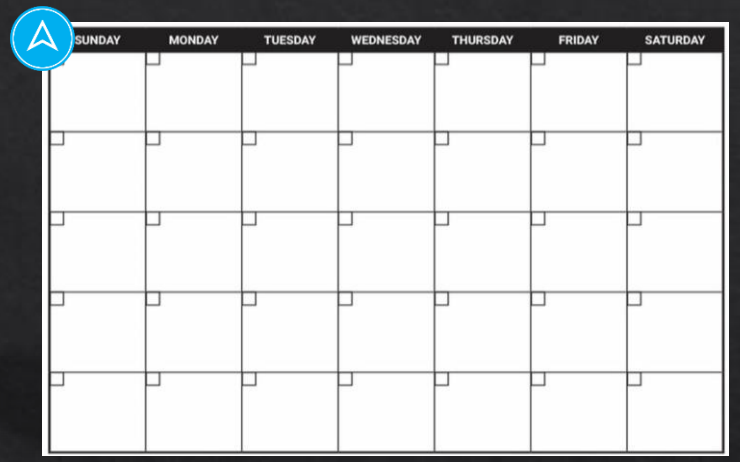
socrative.com – Rooms WLM3PULSESURVEY



TECH ELEVATOR

Announcements

CAMPUS CALENDAR



February 2nd , 2024


TECH
ELEVATOR
Review
WEEK 3 REVIEW
• Module 1 - Wks 1-3 – Checkpoint
• Lecture Review
• Pair Assignment Overview
ELEVATE YOURSELF

TECH ELEVATOR

Review

MODULE 1 – CHECKPOINT

20 MINUTES

Review

REFERENCE TYPES & PRIMITIVE TYPES

TECH ELEVATOR

Review

PRIMITIVE DATA TYPES

Data Type	Size	Description  Value stored on the stack
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

★ Note: **String** data type. Another very common data type is **String**, which is used to store text like **char**, but can be used for more than one character. It isn't a value type, so **String** isn't in the list shown above. However, **String** can often be used like a value type due to special support in the Java language. Note: Strings are immutable.

REFERENCE TYPES

Complex data types, arrays, objects, strings, etc.

Value stored on the Heap

PASS BY VALUE VS. PASS BY REFERENCE

When passing variables around, primitive variables behave differently than variables that reference memory on the heap.

- If a primitive variable is passed to a method and is modified in the method, the value of the original variable will remain the same after the method call.
- If a reference variable is passed to a method and is modified in the method, the value of the original variable will have the modified value after the call. This is because the variable is referring to a location in memory, so the data is retained even after the method call is removed from the stack.

TECH ELEVATOR

Lecture

COLLECTIONS
ARRAYS VS LISTS VS MAPS VS SETS

MAKING THE DECISION: ARRAYS VS LISTS VS MAPS VS SETS

- Use **Arrays** when … you know the maximum number of elements.
- Use **Lists** when … you want something that works like an array, but you don't know the maximum number of elements.
- Use **Maps** when … you have key value pairs.
- Use **Sets** when … you know your data does not contain repeating elements.

PRIMITIVE WRAPPER CLASSES

- Lists and other collections can hold objects.
  - Wait… what if I want a list of ints? Or floats? Or doubles?

- Java has a wrapper class for each primitive data type.

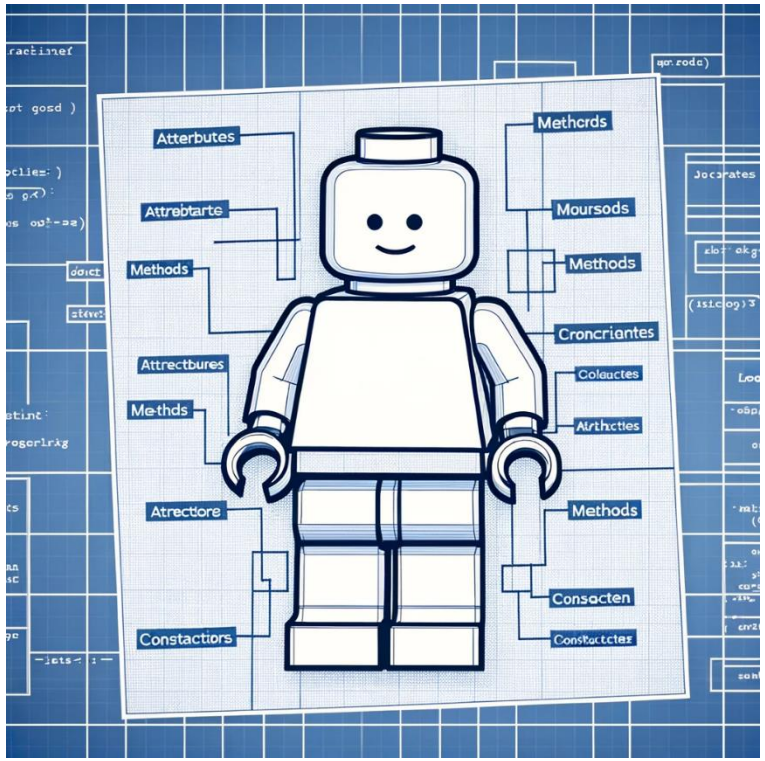
Java Primitive Type	Wrapper Class	Constructor Argument
`byte`	`Byte`	`byte` or `String`
`short`	`Short`	`short` or `String`
`int`	`Integer`	`int` or `String`
`long`	`Long`	`long` or `String`
`float`	`Float`	`float`, `double`, or `String`
`double`	`Double`	`double` or `String`
`char`	`Character`	`char`
`boolean`	`Boolean`	`boolean` or `String`

CLASSES

A **class** is a grouping of variables and methods in a source code file that acts as a *template or blueprint* for creating **objects**.

Class

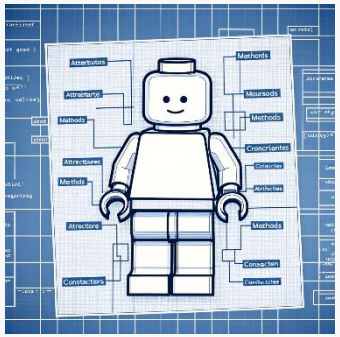
LegoFigure
Properties ⇨ name:String color: String height: int Accessory: String
Methods ⇨ walk() speak() pickUpAccessory()



Lego Figure

CLASSES vs. OBJECTS

When defined in code as a Class, these properties and methods form a blueprint for the creation of Objects in memory. A Class is defined by the code, an object is the “physical” manifestation of a specific instance of that class definition.



ENCAPSULATION

In object-oriented programming, **encapsulation** refers to the bundling of data with the methods that operate on that data, or the restricting/hiding of direct access to some of an object's components.

Lecture

ENCAPSULATION USING INSTANCE VARIABLES

Encapsulation is the concept of hiding or bundling data and controlling access to it.

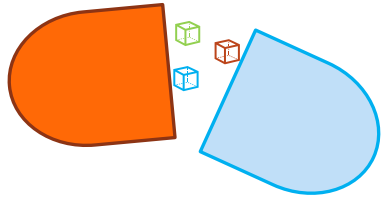
- Letting other code modify data in an instance can be dangerous because it means an instance is not in control of its internal state.
- Hiding code implementation allows other classes to use a class without knowing anything about how it works
- By declaring instance variables **private**, we prevent other code from accessing instance variables directly.
- We use **getters** and **setters** to provide access to internal data to external code.

GOALS OF ENCAPSULATION

Why do we care?

Encapsulation is the concept of hiding data and controlling access to it.

- Encapsulation makes code extendable.
- Encapsulation makes code maintainable.



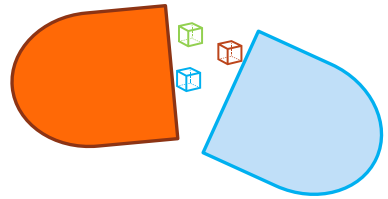
- Encapsulation promotes "loose coupling."
  - A **loosely coupled** system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.

GOALS OF ENCAPSULATION

Why do we care?

Encapsulation is the concept of hiding data and controlling access to it.

- Encapsulation makes code extendable.



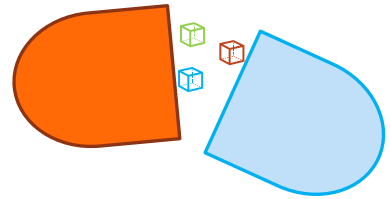
In object-oriented programming (OOP), the term "**extendable**" refers to the ability of a class or software component to be easily extended or modified without making significant changes to its existing code. An extendable design allows developers to add new functionality, properties, or behaviors to a class or component without having to rewrite or alter the core structure of that class.

GOALS OF ENCAPSULATION

Why do we care?

Encapsulation is the concept of hiding data and controlling access to it.

- Encapsulation makes code maintainable.



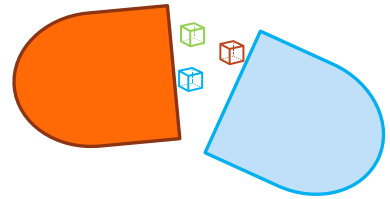
In object-oriented programming (OOP), "**maintainable**" refers to the quality of code and software systems that makes them easy to understand, modify, debug, and extend over time. Maintainability is one of the key attributes of good software design and is crucial for the long-term success and sustainability of a software project.

GOALS OF ENCAPSULATION

Why do we care?

Encapsulation is the concept of hiding data and controlling access to it.

- Encapsulation makes code loose coupling.



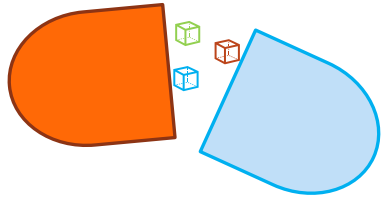
In object-oriented programming (OOP), "**loose coupling**" refers to a design principle that aims to minimize the dependencies between different components or modules of a software system. Loose coupling promotes flexibility, maintainability, and scalability by reducing the interconnections and relationships between classes or modules, making them more independent and less reliant on each other.

GOALS OF ENCAPSULATION

Know these terms

Encapsulation is the concept of hiding data and controlling access to it.

- Encapsulation makes code extendable.
- Encapsulation makes code maintainable.



- Encapsulation promotes "loose coupling."
  - A **loosely coupled** system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.

CONSTRUCTORS

Every class has a **constructor** which is called when an object is being created.

- Constructors are defined similarly to a method but have
  - The same name as the class
  - No return types

```
public SractchPad( ){

}
```

Lecture

CONSTRUCTORS

- To create an object, code must call at least one constructor.
- Java provides a built-in no-argument constructor by default so that each class is not required to provide one to allow basic object creation.
- A class may declare alternate constructors with arguments.
- However, as soon as one or more constructors is explicitly declared in the class, the default constructor is no longer available so if the class needs to allow creation of objects with no arguments, the class must explicitly declare a no-argument constructor to replace the default constructor which is no longer available.

CONSTRUCTORS


```
public SractchPad( ){
    //some code
}
```

```
ScratchPad pad = new  SractchPad( );
```

Here, the **`ScratchPad`** class does not explicitly declare a constructor.

By default, Java provides a no-argument constructor so even though no constructor is explicitly declared for the **`ScratchPad`** class, we can still create a **`ScratchPad`** object using a constructor with no arguments.

CONSTRUCTORS

```
public ScratchPad(String text ){
      //some code
}
```

Here, the `ScratchPad` class declares a constructor which takes a `String` argument.

```
// Valid
ScratchPad pad = new ScratchPad("Testing 123");

// No longer valid
ScratchPad pad = new ScratchPad( );
```

We can now create a `ScratchPad` object using this constructor.

However, because we have declared our own constructor the default no-argument constructor is no longer available so **this is NOT valid**.

CONSTRUCTORS

```
public class ScratchPad {

    public ScratchPad( ){

    }

    public ScratchPad(String text ){
        //some code
    }

}
```

Here, the `ScratchPad` class declares a constructor which takes a `String` argument but also explicitly declares a no-argument constructor to replace the default constructor which is no longer available because the class has declared its own constructor.

```
ScratchPad pad = new  ScratchPad( );
```

Since we have added our own no-argument constructor we can once again create a `ScratchPad` object using a no-argument constructor.

OBJECTS AND THIS

- The `this` keyword is used to refer to the current object.
- We can use `this` to avoid name collisions.

```
public class ScratchPad {

    private String text;
    private String data;

    public ScratchPad(String text, String sampleData ){
        this.text = text;
        data = sampleData;
    }

}
```

The constructor argument `text` has the same name as one of the instance variables.

We use the `this` keyword to differentiate between the instance variable and the constructor argument.

Because the argument being assigned to `data` has a different name, the `this` keyword is not required.

EVERY OBJECT HAS AT LEAST ONE CONSTRUCTOR

WHEN AN OBJECT IS CREATED A CONSTRUCTOR IS CALLED

EVERY TIME YOU USE new, YOU HAVE A NEW OBJECT WITH ITS OWN DATA

Lecture

```
1 Dog jester = new Dog("Jester");
2
3 Dog destroyer = new Dog("Bobby, Destroyer of Curtains");
```

Review

VARIABLE CATEGORIES

- Local variables ✓
- Parameters ✓
- Instance variables ✓
- Class variables ✓

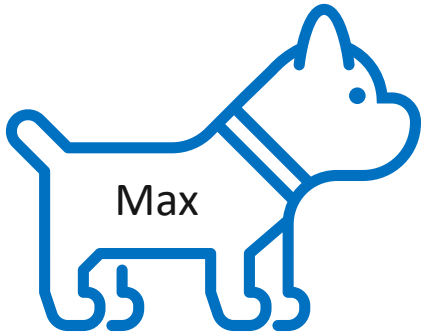
STATIC ATTRIBUTES & METHODS

ANCESTOR: Wolf
dogCount: 2

create instance

name: Giselle
age: 8

create instance



name: Max
age: 4

Dog
public static final String ANCESTOR
private static int dogCount
private String name
private int age

METHODS

Methods

A method is a named block of code that can be called. It can take zero to multiple values and return back a single value.

Methods have **Method Signatures**:

1. Descriptive Names
2. Return Type such as int, long, double, float, object, etc.
3. Input parameters. Parameters are variables that live only in the method.

ANATOMY OF A METHOD

Method signature

access modifier

return type

name

Parameter(s) 0 to n

```
public String calculatePersonalizedMessage( String firstName,  String lastName,  int age )
{
   String fullName = firstName + "  " + lastName;

   String message = fullName + " is currently " + age + " years old.";

   return message;
}
```

body

return keyword / statement

TECH ELEVATOR

METHOD

Review

Calling Code

```
public static void main(String[] args)
{
  String ageMessage =
calcluateMessage ("Neo", 34)

  System.out.println(ageMessage);

}
```

("Neo", 34)

```
public String calcluateMessage (String
name, int age) {
Neo       34
      String message = name + "is " + age + "
            years old!" ;

      return message;
}
```

message

Now ageMessage *equals Neo is 34 years old!*

Console

```
Neo is 34 years old!
```

```
public int addTwoNumbers (int num1, int
num2) {

      int sumTotal = num1 +  num2 ;

      return sumTotal;
}
```

```
public void printConfirmation (int
transactionId) {

      System.out.println("Transaction ID " +
        transactionId + "  has been processed" ;

}
```

Lecture

OVERLOADING METHODS

There are three rules of an overloaded method:

1. Overloaded methods must *have the same name*.
2. Overloaded methods must *differ in the number of parameters, parameter types, or both*.
3. Overloaded methods can have different return types, but that *must not* be the only difference.

```
public String calcluateMessage (String name)
{
   //perform some behavior
   return …
}

public String calcluateMessage (String name, int age)
{
   //perform some behavior
   return …
}
```

Lecture

OVERLOADING METHODS

There are three rules of an overloaded method:

1. Overloaded methods must *have the same name*.
2. Overloaded methods must *differ in the number of parameters, parameter types, or both*.
3. Overloaded methods can have different return types, but that *must not* be the only difference.

```
public String printTotal (int totalSum)
{
   //perform some behavior
   return …
}

public String printTotal (double totalSum)
{
   //perform some behavior
   return …
}
```

Lecture

OVERLOADING CONSTRUCTORS

```
public class ScratchPad {

    public ScratchPad( ){

    }

    public ScratchPad(String text ){
        //some code
    }

}
```

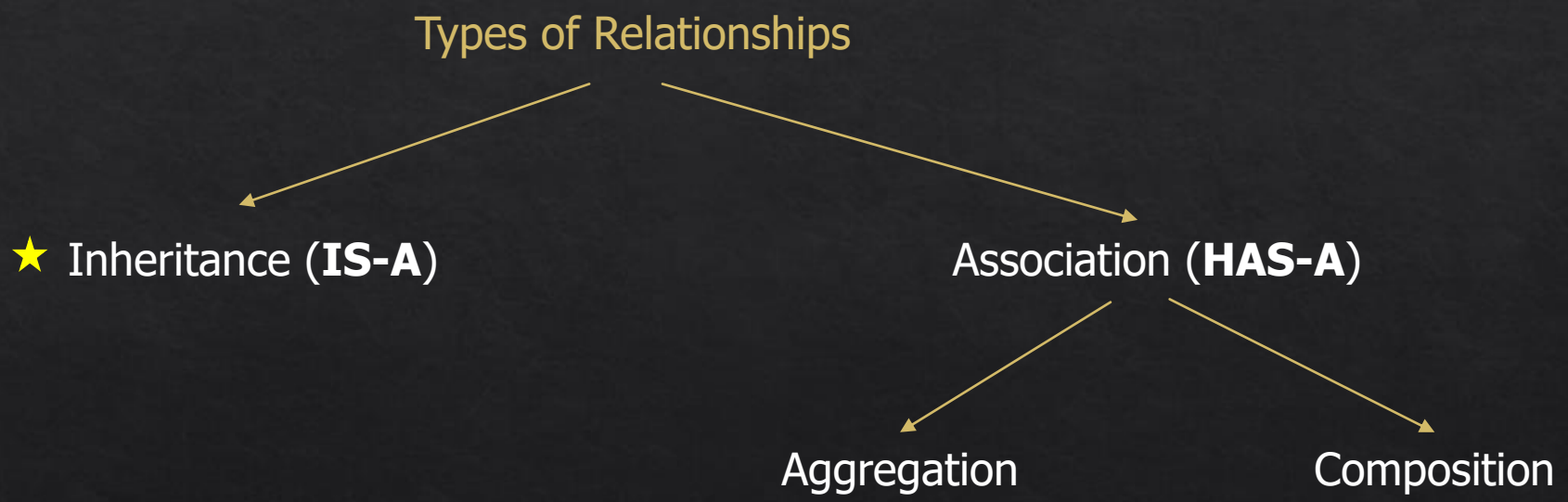
TECH ELEVATOR

Lecture

BREAK

CLASS RELATIONSHIPS

In object-oriented programming, classes→**objects** are related to each other and use common functionality between them.



CLASS RELATIONSHIPS

Plane

Engine

Extends

Association (**HAS-A**)
Composition

Cirrus Vision G2

★ Inheritance (**IS-A**)

Lecture

INHERITANCE

Define a new class based on an existing class; acquiring the properties and behaviors of that existing class.

Lecture

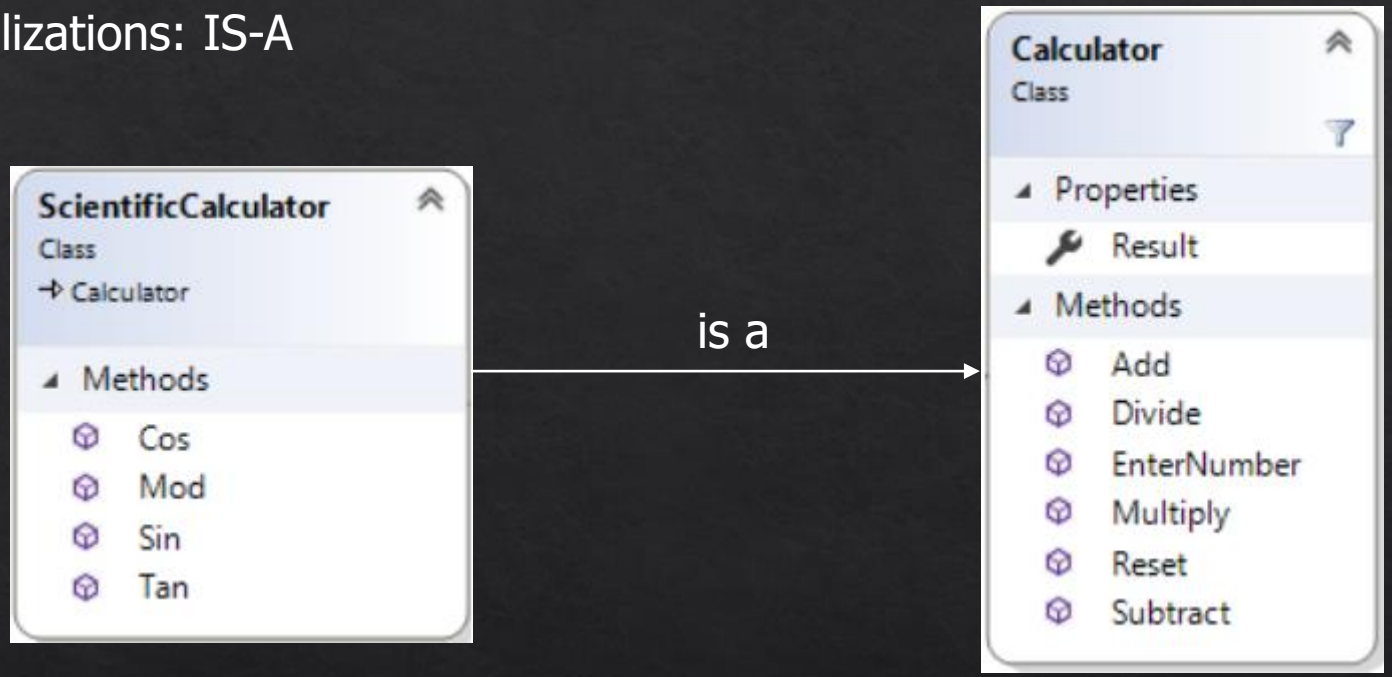
INHERITANCE

If we have multiple classes that are mostly the same, we can create a parent or superclass that has the common or shared parts and child or subclasses that contain the parts that are different.

Inheritance allows child or subclasses to take on the properties and methods defined in a parent or superclass.

INHERITANCE

Specializations: IS-A



- Derived classes are specializations of a base class
- A ReserveAuction or BuyoutAuction ***is a*** specific type of Auction
- A ScientificCalculator is a more specific type of Calculator

Lecture

INHERITANCE

- Allows a class to take on the properties and methods defined in another class.
  - A **subclass** is the derived class that inherits the data and behaviors from another class.
  - A **superclass** is the base class or parent class whose data and behaviors are passed down.
  - All classes are actually subclasses of the **java.lang.Object** class.
  - You may hear superclass referred to as the **parent** class and subclass referred to as the **child** class.
- A class can inherit from another class using the **extends** keyword.
- Subclasses must implement superclass constructors if not using the default constructor (use **super** keyword).
- **private** vs. **protected** access modifiers
  - **protected** acts as **private** to all other classes but every class that extends the class will still have access as if defined with the **public** access modifier.

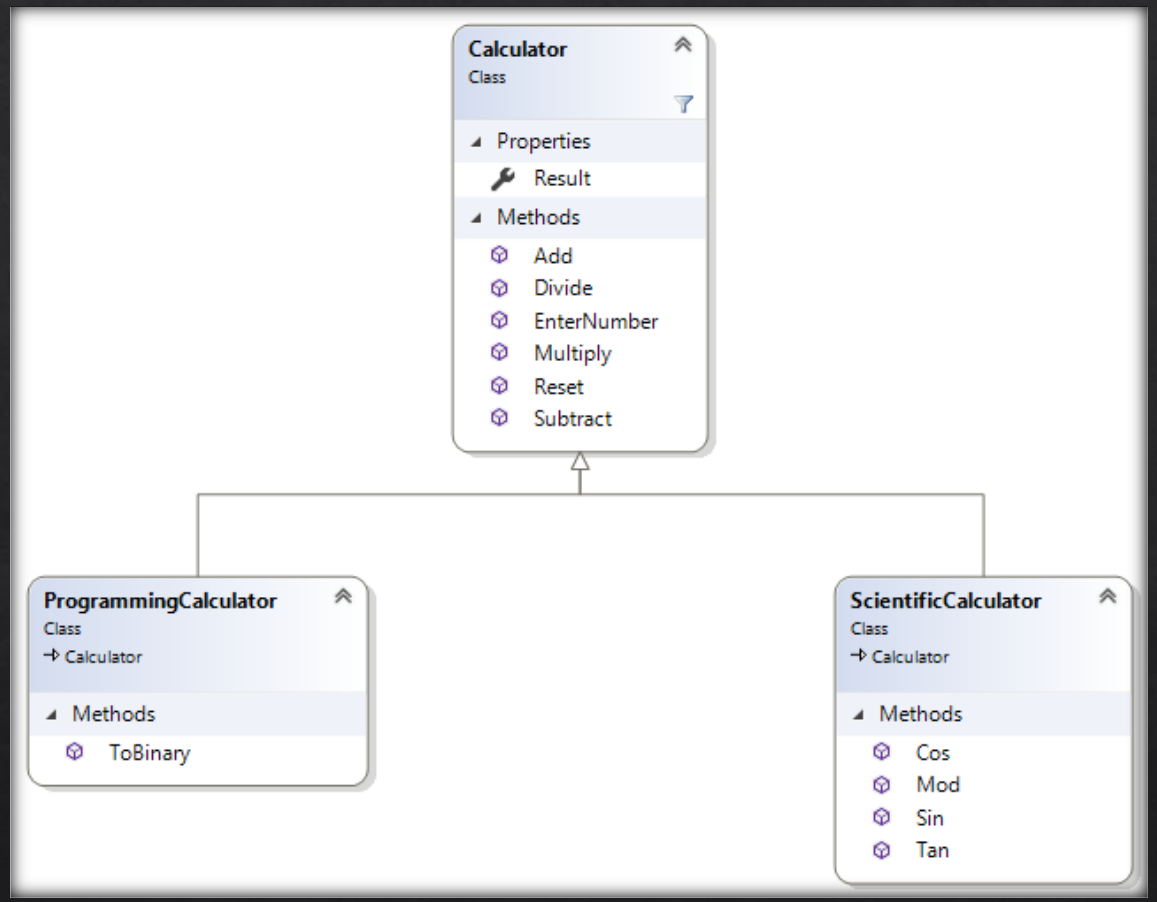
Lecture

INHERITANCE: OVERRIDING METHODS

- A subclass can **override** a method from the superclass by redefining the method.
  - When a subclass method is called, the subclass method will be called if defined, otherwise the superclass method will be.
  - Method **signature must match** the signature being overridden **exactly**.
  - Java provides the `@Override` annotation to make it clear a method overrides the original method.
    - If you use the `@Override` annotation on a method you intend to override, you will get a compiler error if your signature does not match the signature of any signatures in the superclass. This is very useful to ensure your method **WILL** actually override as intended.

- If a subclass overrides a superclass method, that class can always call the superclass method by using the `super.` prefix to access the super version of the method.

EXAMPLES OF INHERITANCE



Lecture

TRANSITIVE

Car is a Vehicle

Honda Civic is a Car

Honda Civic is a Vehicle

Review

4 FUNDAMENTAL PRINCIPLES OF OOP

OOP – Object-Oriented Programming

1. **Encapsulation** - Hiding details of a class behind limited access points

2. **Inheritance** - Similar classes should be able to reuse common code

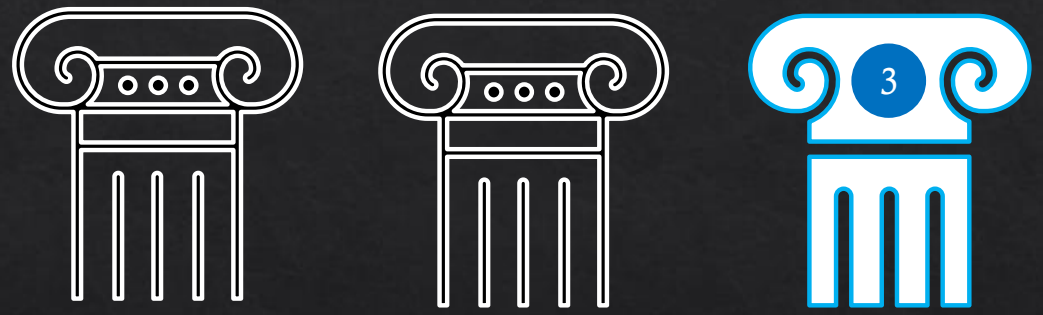
3. **Polymorphism** - Allows different types of objects to be treated as the same type of thing within a program

---

4. **Abstraction** - the principle of handling complexity by hiding unnecessary details from the user. Its goal is to enable the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.

Lecture

POLYMORPHISM



Lecture

POLYMORPHISM

Polymorphism allows different types of objects to be treated as the same type of thing within a program.

Lecture

POLYMORPHISM THROUGH INHERITANCE

In object-oriented programming, **polymorphism** is the idea that something can be assigned a different meaning or usage based on the context it is referred to as. Put another way, different objects can be treated as the same type of thing within a program.

- Polymorphism using inheritance is the concept that any object which is a subclass can be treated as the superclass type.
  - **Auction buyoutAuction = new BuyoutAuction("Super cool thing", 150);**
- Polymorphism can also be implemented using **interfaces**.

Review

POLYMORPHISM THROUGH INHERITANCE

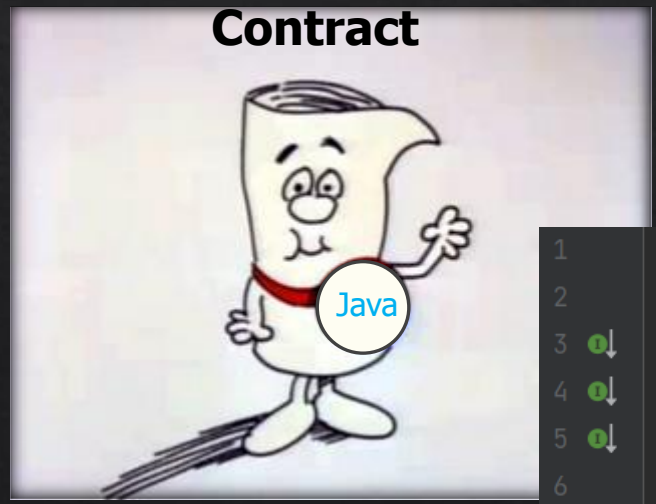
```
FarmAnimal[] farmAnimals = new FarmAnimal[] { new Cow(), new Chicken() };
for (FarmAnimal animal : farmAnimals) {
    String name = animal.getName();
    String sound = animal.getSound();
    System.out.println("Old MacDonald had a farm, ee, ay, ee, ay, oh!");
    System.out.println("And on his farm he had a " + name + ", ee, ay, ee, ay, oh!");
    System.out.println("With a " + sound + " " + sound + " here");
    System.out.println("And a " + sound + " " + sound + " there");
    System.out.println("Here a " + sound + " there a " + sound + " everywhere a " + sound + " " + sound);
    System.out.println();
}
```

Review

INTERFACES

Interfaces define *what something does* but **not** *how* it does it.

Interfaces are contracts that state that classes that fulfill them **must** provide certain methods and properties.



- BEHAVES-AS-A relationships

```
1  package com.techelevator.farm;
2
3  public interface Singable {
4      String getName();
5      String getSound();
6
7  }
```

Review

POLYMORPHISM THROUGH INTERFACES

```
Singable[] singables = new Singable[] {new Cow(), new Chicken(), new Pig(), new Tractor()};

for (Singable singable : singables) {
    String name = singable.getName();
    String sound = singable.getSound();
    System.out.println("Old MacDonald had a farm, ee, ay, ee, ay, oh!");
    System.out.println("And on his farm he had a " + name + ", ee, ay, ee, ay, oh!");
    System.out.println("With a " + sound + " " + sound + " here");
    System.out.println("And a " + sound + " " + sound + " there");
    System.out.println("Here a " + sound + " there a " + sound
            + " everywhere a " + sound + " " + sound);
    System.out.println();
}
```

RESTRICTIONS ON INTERFACES

- **Cannot** be instantiated
- **Cannot** have method bodies*
- Classes **can** implement multiple interfaces
- Classes can use **both** inheritance and interfaces
- Interface methods are always **public**

Lecture

3 FUNDAMENTAL PRINCIPLES OF OOP

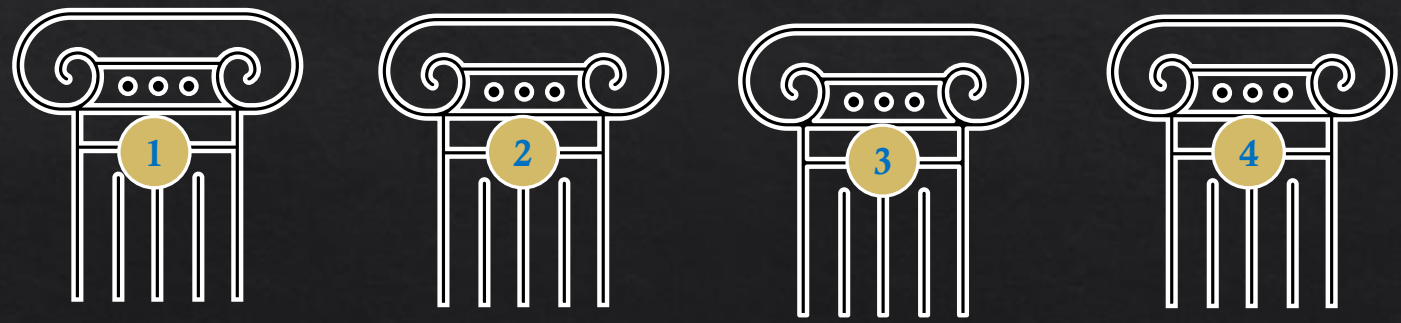
OOP – Object-Oriented Programming

- **Encapsulation** - Hiding details of a class behind limited access points
- **Inheritance** - Similar classes should be able to reuse common code
- **Polymorphism** - Allows different types of objects to be treated as the same type of thing within a program
- **Abstraction** is the principle of handling complexity by hiding unnecessary details from the user. Its goal is to enable the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.
  - Abstraction is sometimes referenced as a fourth principle of Object-Oriented Programming but it is often not included as one of the principles because it motivates the other three principles in one way or another.

4 FUNDAMENTAL PRINCIPLES OF OOP

OOP – Object-Oriented Programming

INTERVIEW QUESTION ALERT!



Encapsulation **Inheritance** **Polymorphism** ***Abstraction***

Lecture

FINAL METHODS & CLASSES

- Making methods `final` means that children can't override what the parent has defined
  - Prevents logic that is integral to the application from being overridden by a poorly behaving subclass
  - Just a design decision that should have a good reason for using
- Making classes `final` means that another class can't inherit from it
  - Again, just a design decision. Should have a good reason for doing it

Lecture

ABSTRACTION & CLASSES/METHODS

- **Abstraction** is the principle of handling complexity by hiding unnecessary details from the user. Its goal is to enable the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.
  - Abstraction is sometimes referenced as a fourth principle of Object-Oriented Programming but it is often not included as one of the principles because it motivates the other three principles in one way or another.

- An **abstract class** is a class that cannot be instantiated. It exists solely for purposes of inheritance and polymorphism.

- An **abstract method/function** is a method/function that does not have an implementation and must be overridden by subclasses.

TECH ELEVATOR

Review

WHAT'S THE DIFFERENCE BETWEEN AN INTERFACE AND AN ABSTRACT CLASS?

INTERVIEW QUESTION ALERT!

Review

ABSTRACT CLASSES VS INTERFACES

- A class can only inherit from one other class, but it can implement any number of interfaces.
- An abstract class can provide implementations for some or all of its methods, while an interface normally doesn't provide any.
- An abstract class can have private methods, while all of an interface's methods are public.

★ The implication of these differences is that an interface provides a lightweight way to use potentially unrelated classes together for a particular purpose, while an abstract class provides a way to share code within a similarly structured set of classes.

Abstract classes and interfaces aren't mutually exclusive, either. It's entirely possible for a class to both extend an abstract class and implement several interfaces.

Review

ACCESS MODIFIERS

Modifier	Same class?	Same package?	Descendant?	Anywhere?
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No
`private`	Yes	No	No	No

Review

SOLID Design Principles

(extended lecture material)

SOLID principles are the design principles that enable us to manage most of the software design problems. **Robert C. Martin** compiled these principles in the 1990s. These principles provide us with ways to move from tightly coupled code and little encapsulation to the desired results of loosely coupled and encapsulated real needs of a business properly. SOLID is an acronym.

“Clean Code”
Robert C. Martin (Uncle Bob)

“The Pragmatic Programmer”
Book by Andy Hunt and Dave Thomas

https://www.c-sharpcorner.com/ - Damodara Naidu

Review

SOLID Design Principles

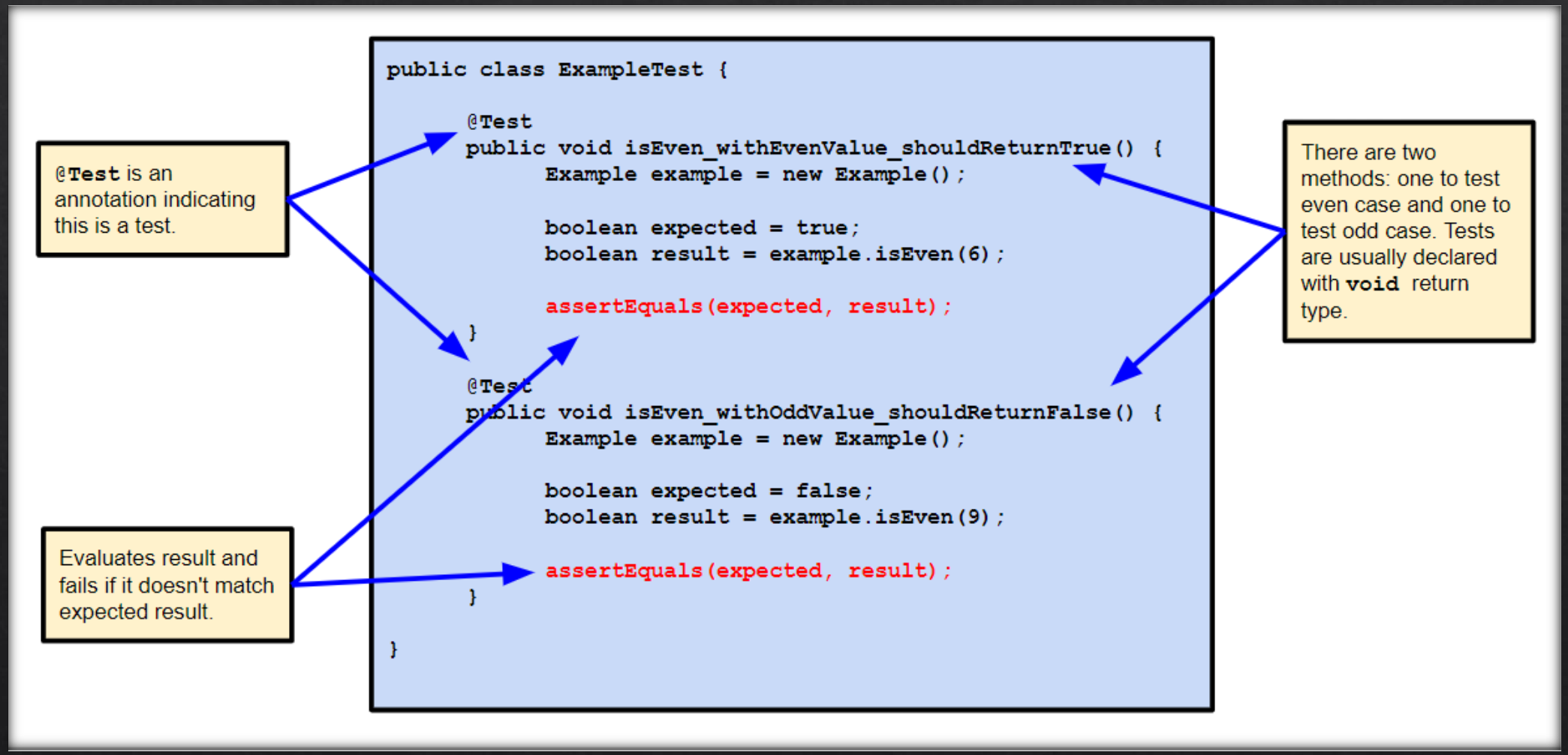
SOLID design principles in C# are basic design principles. SOLID stands for:

- **S**ingle Responsibility Principle (SRP)
- **O**pen closed Principle (OSP)
- **L**iskov substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP).

UNIT TESTING STRUCTURE

- **Arrange**: begin by arranging the conditions of the test, such as setting up test data
- **Act**: perform the action of interest, i.e. the thing we're testing
- **Assert**: validate that the expected outcome occurred by means of an assertion (e.g. a certain value was returned, a file exists, etc.

MORE ABOUT TEST THE TEST CLASS



COMMON ASSERTS

- `assertEquals / assertNotEquals`
- `assertNotNull / assertNull`
- `assertTrue / assertFalse`
- `assertArrayEquals`
- `fail`
- `assertThat`

More info on using asserts:

https://www.baeldung.com/junit-assertions (only JUnit4 portion applies to what we will be doing)
https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html

Lecture

UNIT TEST MUST BE

- Fast
- Reliable / Repeatable
- Independent
- Obvious

Lecture

BEST PRACTICES

- No external dependencies
- Each test should test one concept
- Tests should be of the same quality as production code

Lecture

WHAT TO TEST

- if statements – check both true and false paths
- Loops - test with zero, one, and multiple elements
- Objects – pass in null, normal, and malformed objects

Lecture

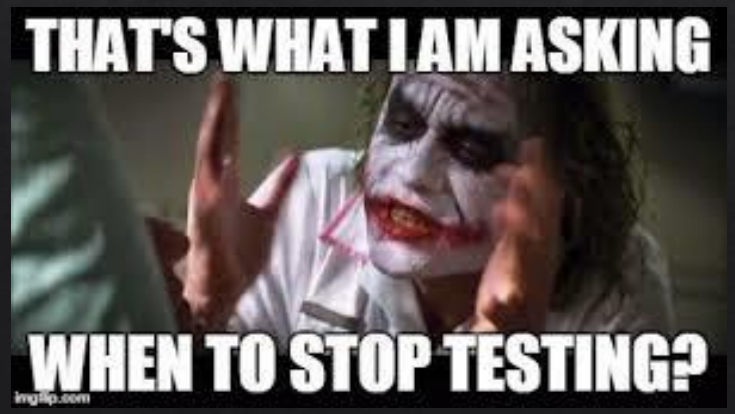
HOW SHOULD UNIT TESTS BE STRUCTURED?

⚠ INTERVIEW QUESTION ALERT!

1. **Arrange**: Set up the code we want to test. (necessary variables and objects, etc.)
2. **Act**: Do some form of an action that we want to verify is correct (call the method being tested, passing any parameters needed)
3. **Assert**: Verify expected results (system under test behaved the way we wanted it to)

WHAT IF WE DEFER TESTING

- Confirmation bias fools us into thinking our code works
- We run out of time for testing
- We forget to test some of the things



Lecture

CODE COVERAGE

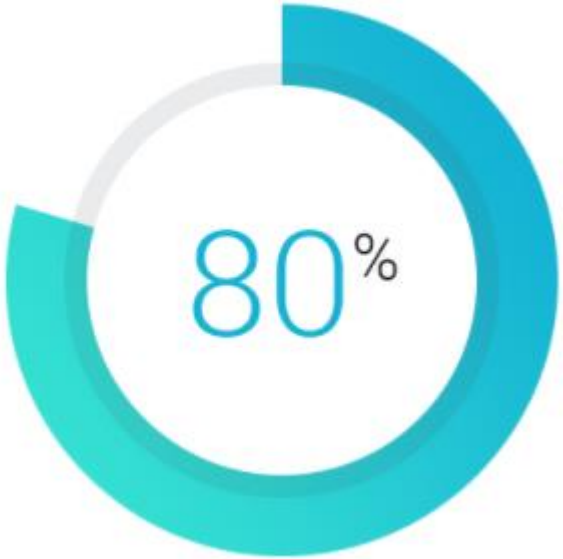
```
public class Circle
{
    public int Radius { get; set; }
    public Point Center { get; set; }

    public double GetLength()
    {
        var result = 2 * 3.1 * Radius;
        return result;
    }

    public double GetSquare()
    {
        var result = 3.14 * Radius * Radius;
        return result;
    }

    public double GetSectorSquare(float angle)
    {
        var result = 3.14 * Radius * Radius * (angle / 360);
        return result;
    }
```

Covered. Tests fail

Covered. Tests pass

Not covered

CODE COVERAGE TARGETS



Lecture

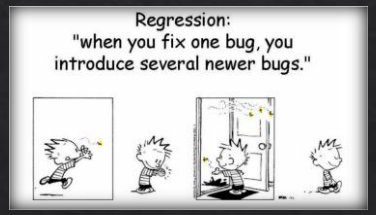
TECH ELEVATOR

Review

EXPLORATORY TESTING

Exploratory Testing explores the functionality of the system looking for defects, missing features, or other opportunities for improvement. Almost always manual.

REGRESSION TESTING



REGRESSION TESTING is defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features. Validates that existing functionality continues to operate as expected.

★ UNIT TESTING

Unit Testing is low level of testing performed by programmers that validates individual "units" of code function as intended by the programmer. Always automated.

★ INTEGRATION TESTING

Integration Testing is a broad category of tests that validate the integration between units of code or code and outside dependencies such as databases or network resources.

ACCEPTANCE TESTING

Acceptance Testing is performed from the perspective of a user of the system to verify that the functionality of the system satisfies user needs.

INTERVIEW QUESTION ALERT!

Lecture

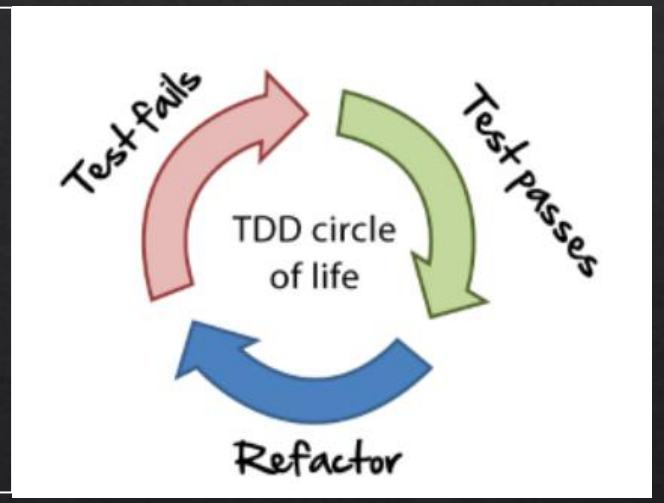
RECAP

TEST DRIVEN DEVELOPMENT

Review

Test Driven Development is a mode of development where we start by writing tests for the improvements or fixes we're about to make.

Once the tests are in place, we change the application and verify that our tests now pass.



Lecture

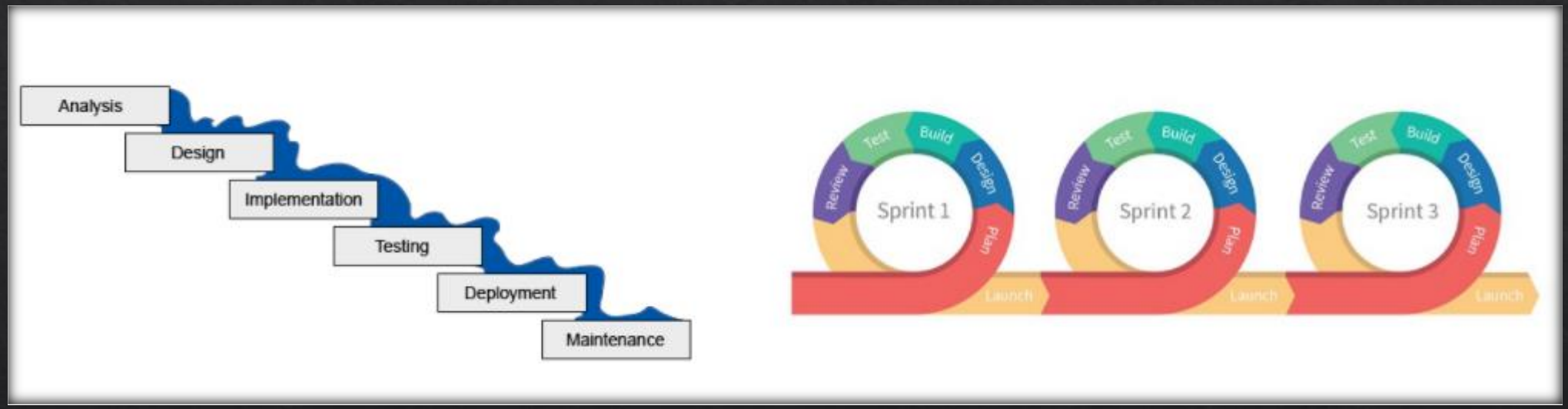
SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

Is a process for planning, creating, testing, and deploying software.

INTERVIEW QUESTION ALERT!

WATERFALL vs AGILE

SDLC METHODOLOGIES



Linear Sequential Life Cycle Model

Continuous iteration of development and testing in the software development process.

⚠ INTERVIEW QUESTION ALERT!

Lecture

AGILE DEVELOPMENT

Daily Scrum Meeting

- Time box (15 min)
- Same place
- Same time
- Facilitated by Scrum Master
- Full team presence
- Focus on 3 questions

3 Main Questions:

1. What did I do yesterday?
2. What will I do today?
3. What's in my way?

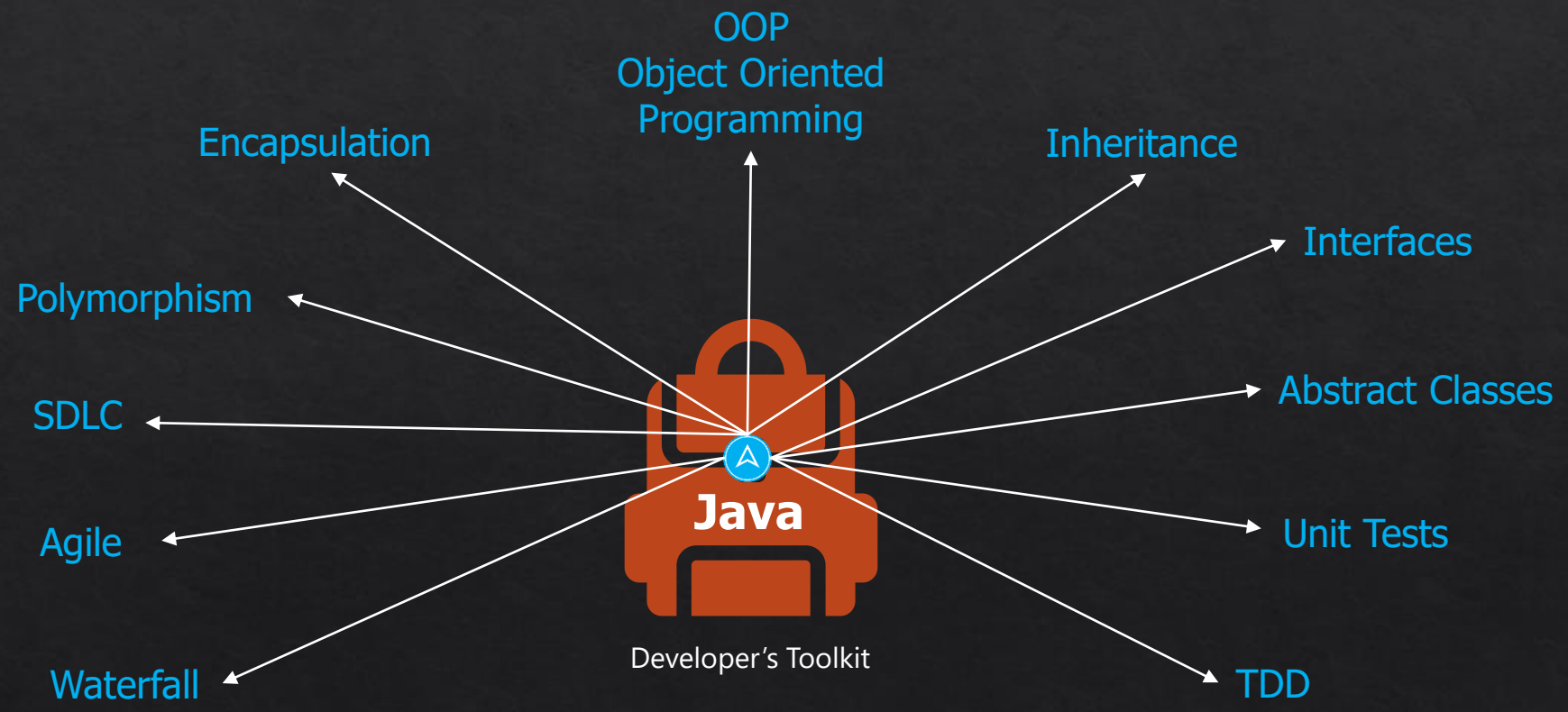
Lecture

TECH ELEVATOR

Module 1

WEEK 3 REVIEW

New Tools! Let's Code!



Exercise HW

TECH ELEVATOR

PAIR PROJECT

DUE MONDAY 11:59PM

Exercise HW