

Java, Maven, Git

= questions/confusion

What are the 2 steps to running Java code?

- 1. **compile** the source code into bytecode
- 2. **execute** the resulting bytecode
- This is Java's method of **packaging & distributing** code (which is a crucial part of software development)
- Languages that have a built-in code/program that reads, parses, and interprets your written code for execution
- This allows the same source code to run on diff platforms & processors...
* however, it is slower.
- Ex: Python, Javascript

What are compiled languages?

- There is a compiler that parses & translates src code into machine executable code
 - * "compiler" is not a universal structure for use across different machines -- the compiling must be done for the specific **ISA (instruction set architecture)** used by the machine
- Compiled languages run faster, but a machine-specific version
* needs to be created for each machine
- "lower-level"; speaks directly to the computer rather than to an interpreter... this is why its faster.
- Ex: C, C++, Rust

What was Java's big idea?

- to take the best of both worlds by compiling source code to the "machine code" of a virtual machine
 - * this is where **JVM (Java Virtual Machine)** comes in
- they also provide machine specific implementations of the virtual machine
- Pro: write (code) once, run everywhere
- Con: some performance loss
- The process of translating a program from source code into a language that a machine can more easily understand.



- The JDK includes a Java compiler that converts your Java source code into bytecode, which is formatted in a '.class' file.
- bytecode wouldn't make sense to us: its like "CA FE DO 11 23 45" type stuff.

What does "executing" mean for a Java program?

- How is it done?

- The process of feeding a compiled (bytecode) program into a machine that can follow the instructions.
- the JVM reads the bytecode & executes it
 - SOURCE code is not even needed to execute the program since it has been converted/duplicated as a bytecode file.
- Since the bytecode is not specific to any machine, it can be read and executed on any computer!

- Compiling & Executing Large Projects -

What do large projects usually have?

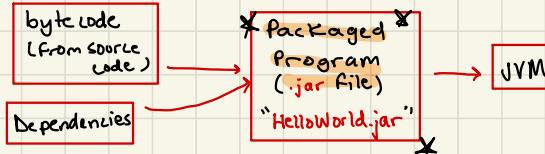
What is a dependency?

How is a large project compiled?

How is it executed?

So what does the final, executable program look like?

- ① Hundreds of .java source files to compile
- ② External dependencies that must be imported
 - A "3rd party" library containing code that you didn't write yourself, but is used in your project
 - Examples that we will use: JavaFX, JUnit
 - they are usually in a .jar file format, which is just a bunch of ".class" files Archived into one.
- all of the source code files and the dependency files get passed through the JDK for compilation into bytecode.
- all dependencies & compiled bytecode are passed through the JVM for execution
- all compiled .class files AND all dependent .jar files must be present at runtime.
 - Luckily, all of these files can be packaged together into a single, distributable .jar file



- all files packaged into 1 .jar file
- source code does NOT have to be stored/included in this jar file, since we already have the corresponding bytecode.

How do you compile source code from the command line? (like in the terminal)

```
$ javac -classpath C:\proj\deps;C:\proj\junit.jar HelloWorldA.java HelloWorldB.java
```

- ① this line tells the computer to compile a Java program.
- ② Listing dependencies to use when compiling (they are listed by their location on your computer)
 - multiple dependencies are separated with semicolons ✘
 - dependencies can be .jar or .zip files, or paths/directories containing class or jara files.
- ③ Listing all of the .java src files to be compiled into .class files
 - multiple files separated with spaces ✘
 - must all be .java

How do you execute code (bytecode) from the command line?

```
$ java -classpath C:\proj\deps;C:\proj\junit.jar com.comp301.lec01.ex01.HelloWorld arg01
```

- ① This line tells the computer to execute a Java program.
- ② List of bytecode files to include in execution (listed by their location on your computer)
 - multiple bytecode locations are separated with semicolons ✘
 - can be .jar or .zip files, or paths/directories containing class files
- ③ Full name of the class to be executed
 - the specified class must define a "main()" function
- ④ the command line arguments

- Packages and Imports -

What are java packages?

- a 'namespace' that organizes a set of related classes & interfaces.
- conceptually, can think of packages as being similar to different folders on your computer.

So what are they for?

- for organizing class files into different "units"
- can put related class files in the same package, & unrelated ones in diff packages
- classes exist in packages
- packages are basically an organizational unit
- NOT the same as concept of "packaging" code into a .jar file.

How do you put a class in a certain package?

What does each part of the name mean?

- packages' names imply hierarchy ; don't just name them randomly
- put a line like this at the top of the class file:
`package com.comp301.lec01.ex01;`
- Each part of the package name is a subFolder within the package
- class files are placed into the bottom-most folder associated with their package name
- Packages are associated with a particular folder path on disk ; the package name is essentially a file path.

Breakdown of this package name?

HelloWorld.java (source code)

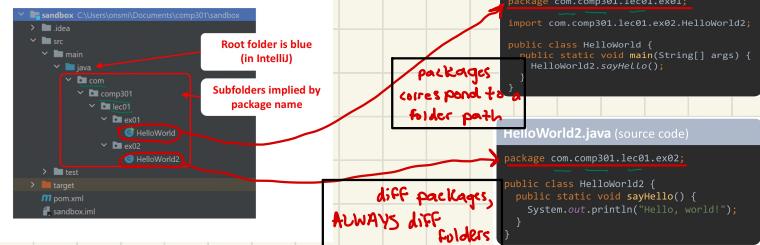
```
package com.comp301.lec01.ex01;
```

→ the class is in the "ex01" folder, which is in the "lec01" folder, which is in the "comp301" folder which is in the "com" folder.

→ HOWEVER, the each indiv. folders are NOT packages themselves. There's only

1 package, which is the entire line

→ several files can share a common prefix folder path (aka package name):



How do you reference across packages?

→ does every file always have its own unique package?

→ call a package of a different file by writing out its name, the name of the file w/ that package, & the method (in that file) that you are wanting to use:

HelloWorld.java

```
package com.comp301.lec01.ex01;
```

```
public class HelloWorld {
```

```
    public static void main(String[] args) {
```

```
        com.comp301.lec01.ex02.HelloWorld2.sayHello();
```

```
    }
```

↳ calling the file "HelloWorld2.java" via its package name.

→ referencing is inefficient because it's a super long line to type & you have to type it EVERY SINGLE TIME you want to use a method from another src code file.

→ Imports! Import statement only needs to go once at the top of the current file... then you can call methods from the referenced file whenever you want:

HelloWorld.java

```
package com.comp301.lec01.ex01;
```

```
import com.comp301.lec01.ex02.HelloWorld2;
```

```
public class HelloWorld {
```

```
    ...
```

???

The import statement

- The Java Build Process -

What is the "build process"?

→ In software engineering, it is the act of converting a project's source material (source code, image files, raw data, etc.) into a shippable software product.

↳ meaning that the output is packaged up & ready to be sent to the customer.

→ essentially, the steps you take to clean up & get a program ready for publishing

→ Build process tasks vary from project to project and from language to language

What are common build process tasks for Java projects?

① **clean** : delete leftover temporary files from the project folder

② **compile** : source code → bytecode

③ **test** : Run the unit tests for the project

④ **package** : pack the compiled code into a distributable format

⑤ **verify** : check that the packaged output meets quality criteria

⑥ **site** : Generate documentation for the produced code - notes or other documents you might make.

⑦ **deploy** : Send the packaged output to customers

What are "build automation tools"?

→ softwares or programs or systems used to make all the steps of "building" easier.

→ Different programming languages use different BATs

→ Popular BATs for Java:

• Ant

• Gradle

• Maven

what we will use
in COMP 301

Definitions of Maven terminology?

→ **pom.xml** : the Main project configuration file

→ specifies various settings for the project

→ typically placed in the root of the project folder.

→ written in XML, which is a generalized form of HTML.

→ **Dependency** : external module or library that your project uses.

→ **Archetype** : A template for creating new Maven projects.

→ **Artifact** : the packaged output file(s) produced by the project.

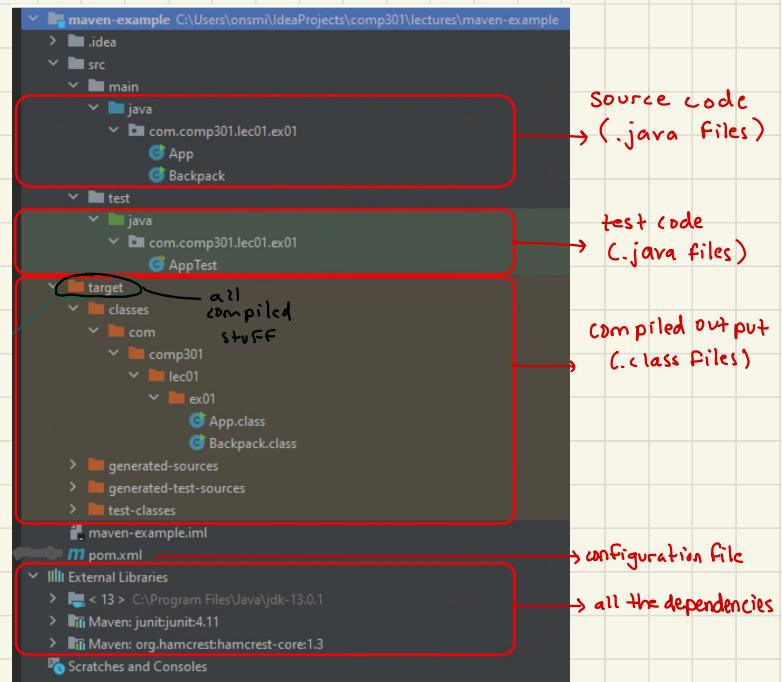
→ **Lifecycle** : A configurable build process task ; All of the tasks that Maven is automating. The "build tasks".

→ To go with reasonable, conventional defaults unless you tell it to do otherwise.

• What is the point of branches?

What is the idea behind Maven?

Example of a Maven project directory?



→ All (COMP 301) assignments will follow this structure

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.comp301.lec01.ex01</groupId>
  <artifactId>maven-example</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>maven-example</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>...</build>
</project>
```

Project version & artifact info

Project name

Java version and other properties of the project

Dependencies

Lifecycle configuration

these lines tell us that the project is depending on the JUnit library, & that we need version 4 of that library.

→ Knowing how to write all of this stuff isn't that important; Maven does it for you.

→ all you have to do is copy & paste the "Maven dependency snippet" (that many public Java libraries usually provide) into your pom.xml file.

→ When you add them, Maven will automatically download the correct .jar file link it during compilation/execution.

How do you add dependencies to your project?

- The Git version control system -

What is "version control"?

version
control
system

Why is having a VCS important?

What is Git?

What is a commit?

How are commits stored?

Where are commits stored?

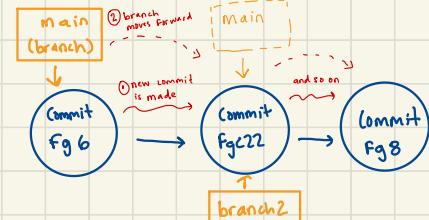
What is a repository?

What is a branch?

Can you have more than one branch?

What is the checkout operation?

- The practice of tracking changes made to a codebase
- i.e., periodically capturing "snapshots" of the code, and archiving all snapshots in case you need to refer back to them
- all software companies have a version control system in place.
- Especially useful when there are multiple developers working on one source code.
- Useful for when you need to backtrack your work.
- the most well-known & widely used VCS
- A "snapshot" of the files in a codebase at a point in time.
- Basically every time you make a new commit, a version of your code is saved. Kind of like looking at "revision history" on Google docs.
- When one is created, it is given a unique identifier generated based on the content of the files stored inside
 - The identifier is similar to a Hash Value (for ex., "f22e9a3")
 - identifier is used to reference the commit.
- Every commit has a parent commit, thus forming a graph
- You choose when you want to make a new commit... whenever you have made enough changes & want to save them
- Everywhere; both locally & in Github. If someone clones your project, they will get all of your commits too
- A "storage unit" for tracking & storing commits related to a project.
- A remote repository is stored on Github, and a local one on your computer.
- A movable label that points to a particular commit.
- Everytime a new commit is made, the current branch is moved forward to point to the newest commit.
- Yes, new branches can be created
- multiple branches can point at the same commit
- "`$ git branch branch2`" to create a new branch
- "`$ git checkout (branch name)`"
- changes which branch you are working on (a.k.a., the 'current' branch; the branch that will move forward with each new commit). Checkout a branch in order to make it your new 'current' branch.



What is staging?

- selecting a file (in its most current version) & adding it to the staging area because you want it to go in the next commit.
- Basically, a singular "commit" can save as many files as you want it to, but you can only create the new commit once, with one click.
 - Before making a commit, you add every file (to which you have made changes) to a little waiting room (called the "staging area") -- This process is called staging.
- THEN, you "make your commit", & it takes out everything in the waiting room and puts it in the commit, thus incorporating the staged changes.

What is the command syntax for:

• staging?

→ `$ git add "file1.txt" "file2.txt"`
picking the files you want to commit

→ to stage every file in the project:

`$ git add .`

• committing?

→ `$ git commit -m "commit message"`

• every commit must have a message before being sent. You should use the message to briefly describe the new changes that were made.

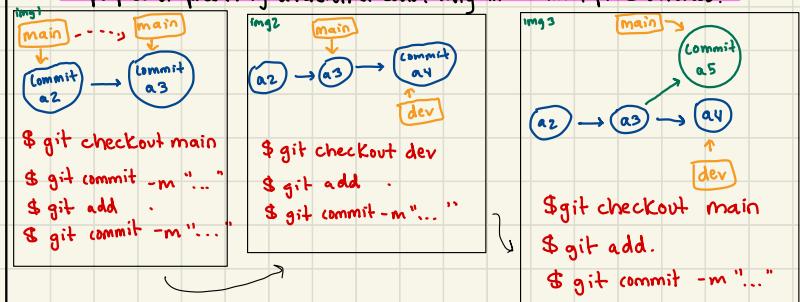
Branching and Merging - (Git operations)

What is branching?

Why is it useful?

- 2 branches pointing at the same commit can create diverging versions of the code.
- since only 1 selected branch can move forward with every commit.
- It allows you to diverge from the main line of development & continue to do work without messing up the main line.
- you can attempt new stuff (like adding features or fixing bugs) w/o having to worry about ruining your main code, since a version of it is saved under a different branch.
- Why do you need multiple branches to do this?? Doesn't "committing" already serve the purpose of preserving a version of code? Why involve multiple branches?

Example of branching?



→ by switching back to an old branch & then creating another version of the code, with different changes committed, you have branched! Notice how image 3 starts to look like a tree with branches.

Next: Fetching, pulling, pushing

Motivating the OO (object-oriented) way

What is the non object oriented approach?

- All functions are static
- All of the code for a program is in just one file/class... a.k.a you do everything in "public static void main { ... }"; no external classes are called; very elementary?
- All variables are either declared locally or passed in as parameters.
- use static helper functions within the main class (like we did in UD assignments)
 - all of these functions act as a library of functions used by our application.
- non object-oriented; global; can be called by anybody
- a special function signature that Java uses to identify where to start the program
- so only one class in a program will have it?
- any external functions, like Scanner, .equals, etc. are all objects; it's basically useless to try to create a program without any objects

String triangle_category (double ax, double ay, double bx, double by, double cx, double cy) {
 ↘ ↗ ↘
 the value type name of Function parameters
 that the function will produce/return

- ax, ay, bx, by, cx, cy are all double values that specify the x- and y-co-ords of a triangle... in the non-OO example, we created a bunch of static helper methods to calculate perimeter, area & triangle type (isosceles etc.) for a given set of co-ords.
 - in non-OO, these vals were provided by us as parameters.
 - Also on us to manually verify that each set of 6 vals actually represent a triangle

What is the idea behind OO - programming?

- It flips this relationship between the input & the functions performed on it.
- Designs/creates a language that allows us to collect data together as an abstraction... and encapsulate this abstraction into a piece of software that provides us a much easier way to work with the data w/o having to know any of the internals & specifics of how that data is interpreted, maintained, & used.
- for ex, if you have a ton of data of triangle co-ords & you want to use it to do something else (like in a completely unrelated project)... then it makes WAY more sense to create a separate class containing all of these calculations & methods, which we can then utilize across/in any other file by calling it... than to type out all of the calculations all over again in your new project.

idea behind OO-programming (con't.)

- Formalizes the collective meaning of these pieces of info, as an abstraction.
- The abstraction provides a means to query properties, invoke "behavior", & save objects of that type.

- The Steps to OO-programming -

①

Name the abstraction

How are Java OO programs structured?

→ In Java; create a class corresponding to the abstraction name.

- i.e., what is the 'thing' we are trying to abstract
- ex: Triangles

red = referring to Triangles.java example

→ so, a new.java file named after the abstraction
`Triangle.java` `public class Triangle {`

- 1 class with a main method, where we do the programming
- several other classes that define objects that are used by that program.

② declare its fields

How do you declare the fields?

→ we want to collect the data that defines the abstraction; the fields of the object/abstraction are pieces of info that collectively define it.

→ `ax, bx, cx, ay, by, cy ... all double values`

→ they follow variable naming rules of "type name;"

→ BUT, they are not local variables. They shouldn't be inside any function or method, rather floating off by themselves.

→ declare them all at the top of the class body (for organization)

`public class Triangle {`

`double ax;`
`double ay;`
`double bx;`
`double by;`
`double cx;`
`double cy;`

} these are the field names

→ [name of object instance] + . + [field name]

`int num = testtriangle.length;`

How do you access an object's field value in the main method?

- **object**: the thing being abstracted (`Triangle`, for ex) ... there is only 1 object
- **instance**: every new version of the object that is created by the constructor is an "instance" of the class (`"Triangle test2"` is an instance of the `Triangle` class)
- **method**: Javaspeak for "function" ... they are synonyms.

TERMINOLOGY:

③ define a constructor

What is a constructor?

→ Job of constructor: fill new object with values in its fields.

→ A special type of thing (?) that creates & initializes new instances of the object

→ Although it somewhat acts like a method, it is NOT a method/function

→ Constructor is run when we ask to make a new object (**triangle**), and fills in the Fields of the new instance.

→ they are NOT considered members of the class

- ① fields ← the collection of fields &
- ② methods methods make up the "class members"

→ differs from normal method ; specific form/rules to follow :

- the name of the constructor must match the class name
- Does not have any return value or return type, because it doesn't return anything... just used to create instances/objects
- cannot be called, the way methods can

→ whenever we run a line of Java code to create a new instance, Java will set aside the memory needed to make it (in the heap)

→ It knows how much memory to set aside based on the Fields of the constructor

• sets aside enough memory to store 6 real numbers.

→ After creating the piece of memory, Java starts the constructor.

→ a special keyword inside the constructor that refers to the new object to be initialized

→ it's essentially a memory pointer /reference to the memory that Java set aside for the new object... so that computer knows where to send the data that is passed through the class fields.

→ Can think of it as the current object.

① **new** Keyword invokes the constructor & effectively allocates space for the new object.

② the constructor fills the object fields of the current object with the keyword **this** (like **this.a = 3**) ... "this" points to the space created by **new**, and the object data/field info is sent there.

→ by passing them in as parameters, similar to what we do with methods

How do you give constructor info to fill its fields?

What are the "members" of a class?

How do you declare a constructor (inside a class)?

How does Java create memory for instances?

What is "this"?

Example of declaring the constructor?

→ in the class body;

① "public" meaning that any file can use the constructor

② Name of constructor (same as name of class)

③ all the values the constructor takes in, to build the object; the parameters.

④ "this" points to the new piece of memory that was just created for this instance, and fills the Fields (ax, ay, etc.)

With the data passed in by us through the parameters (x_1, y_1, \dots)

• FORMAT: `this.[name of field] = [name of parameter];`

→ using the Keyword "new"

→ new invokes the constructor } and allocates the memory for this new object.

→ In the main method;

`Triangle test1 = new Triangle (5.0, 2.5, 3.0, 6.0, 6.0, 4.0);`

the values of x_1, y_1, x_2, \dots ; the values of the parameters, set by us (the user)

Example of calling the constructor?

When is "this" implied (and can be omitted)?

→ If the name of the field is not already the name of any local parameters or variables in the function, then Java automatically treats it as if a field of "this" object ... so "this." is not necessary when setting that field:

`public Triangle (double x1, double y1, double x2, double y2, double y3, double y3) {`

`this.ax = x1; } → ax = x1;`

`this.ay = y1; } → ay = y1;`

`this.bx = x2; } → bx = x2;`

`... }`

→ BUT, if the parameter names of the constructor match the field names, then we can't use implied this because its unclear:

`public Triangle (double ax, double bx, double cx, double ay, double by, double cy) {`

$\begin{matrix} ax &= & x_1; \\ ay &= & y_1; \\ bx &= & x_2; \\ ... & & 3 \end{matrix}$

the computer will assume you are referring to the parameter ax, not the field in the constructor, so it will not fill (or at all reference) the field ax like you are wanting it to.

④

Define instance methods

What are instance methods?

- attaching "behaviors" to the object, that make it possible for the object to execute actions or compute things about itself, and return an answer.
- Functions/procedures that depend on the specific instance
- coded inside the class body
- These functions execute instructions to return a value based on info from the fields, for ex, a method that returns a string with a category name for a triangle object (scalene, isosceles, equilateral) after doing computations with the co-ord values
- Without a "static" Keyword because they aren't global/general... they are specific to each instance of an object/class
- called in main method with the "..." operator;
`reference.method();` or
`this. field. method();` or
`field. method();`

How do you declare one?

How do you call one?

static versus instance methods?

- Static: general, public methods
 - referenced through the CLASS (called) name (if/when being used in a diff class)
 - coded in main method
 - static double point_distance (double x1, double x2, double y1, double y2) {
 return Math.sqrt(((x1-x2)*(x1-x2)) + ((y1-y2)*(y1-y2)));}
 - This is an example of a static method being called
- Instance: only make sense if referenced through a particular instance
- referenced through the instance (called) object's name
 - coded in the class body
- ```
public double area () {
 double side_ab = TriangleMain.point_distance(ax, ay, bx, by);
 double side_bc = TriangleMain.point_distance(bx, by, cx, cy);
 double side_ca = TriangleMain.point_distance(cx, cy, ax, ay);

 double s = (side_ab + side_bc + side_ca)/2.0;
 return Math.sqrt(s * (s-side_ab) * (s-side_bc) * (s-side_ca));
}
```
- calling (in main method);  
`System.out.println(test1.area());`

## Summary: classes and objects -

What are instance fields & methods?

- Fundamental units of abstraction in Java.
- methods in a class<sup>body</sup> that are used to fill Fields with values specific to every instance of the class object
  - every object contains the same fields, but the instance methods are used to derive & assign the specific values for each field in a given instance.
  - for ex; triangle height, area, category, etc.

What are class fields and methods?

- fields & methods in the class body that are not associated with any particular instance
- they define values & helper methods that are associated with the class/ abstraction as a whole ; i.e. one constant value for every instance of the object.
- distinguished by the static keyword. (other than that, declared in the same way as instance fields.)
- Ex: named constants that will be used inside the class
  - convention is to declare static fields @ in all caps , and ⑥ with the "final" keyword

What is the final keyword?

What are objects?

- fields that are marked as final can never be reassigned after the constructor has given them their initial value.
- each object is an instance of the class
  - (ignore my mixing up terminology before, it doesn't really matter that much as long as you understand the concept of an object type (Triangle) (aka the class), and of all the separate instances of the class (test1, test2, Avitriangle, etc.)
- an object is a collection of named fields that represent information about that object.
- the "state" of an object is reflected by the values currently assigned to those fields.
- the "design" of an object (aka the decision of what fields to include in an object) reflects its purpose. (how the object will be used.)
- When we type "new", we get back an object called new.
- Every time we call new; we get a different new object which the constructor then fills with values.

Summary: how does "new" work?

What is a physical analogy  
for classes & objects?

→ CLASSES = FACTORIES :

- blueprint for the object = ★ instance fields (define its data)
- ★ instance methods (define its behavior)
- Factory's facilities & maintenance = ★ class methods
- ★ class fields

→ OBJECTS = WHAT THE FACTORY BUILDS

Summary of how to  
code each thing!

|                                                                    |                                                                                                                                               |
|--------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| Instance fields<br>(each instance gets one)                        | <pre>public class Point {<br/>    private int x;<br/>    private int y;</pre>                                                                 |
| Class fields<br>(entire class shares one)                          | <pre>private static final double EPSILON = 0.001;</pre>                                                                                       |
| Constructors                                                       | <pre>public Point(int x, int y) {<br/>    this.x = x;<br/>    this.y = y;<br/>}</pre>                                                         |
| Instance methods<br>(called on an instance,<br>has access to this) | <pre>public double distanceTo(Point other) {<br/>    return Point.distance(this, other);<br/>}</pre>                                          |
| Class methods<br>(called on the class,<br>no access to this)       | <pre>public static double distance(Point a, Point b) {<br/>    return Math.sqrt(Math.pow(a.y - b.y, 2) + Math.pow(a.x - b.x, 2));<br/>}</pre> |

## Encapsulation

How do you decide what fields to make when designing a class?

- choose fields that fundamentally identify the object
  - the smallest set of info that you can use
- Avoid redundant fields & fields that have relationships to each other that must be specifically maintained.
- minimizes amount of memory used for the object
- reduces / allows you to avoid a lot of bugs.
- EX: The most efficient way to define a square in just 2 fields:
  - ① co-ords of the lower left corner
  - ② a value representing the width (which is = to height)
- the concept of bundling data (a.k.a. fields) together with the operations (methods) performed on that data.
- sometimes also called "information hiding."
- Shield an object's internals from the rest of the program, in order to:
  - prevent instance fields from accidentally being changed.
  - be able to refactor internal code without breaking external code.
- using the "private" access modifier

What is the 1<sup>st</sup> principle of encapsulation?

What is the 1<sup>st</sup> principle of encapsulation?

How is this done?

What is the 2<sup>nd</sup> principle of encapsulation?

How is this done?

What are access modifiers?

What are the 4 access modifiers in Java?

- most encapsulated / protected
- least encapsulated
- ↑
- private : member is only accessible (able to be called) from inside the class body... private fields are "completely encapsulated" in their class.
  - protected : member is only accessible from inside the class and subclasses.
  - default : member accessible from anywhere inside the package
  - public : member accessible from anywhere.
- "private" and "public" are used most often. The other 2 are generally for special cases.
  - if no modifier is specified, assumed to be "default"

- What are getter methods?
- Why are they useful?
- According to encapsulation, all fields in a class should be marked private!! so that there is no risk of their values being accidentally changed or manipulated by code in other classes.
  - A method (in a class, usually) that is public and is used to retrieve the value/data from a particular field in the class.
  - Since class fields are private, if (in main method or other classes) you wanted to retrieve and use the value of some field of an object, you cannot directly get to it.
  - By having public getter methods obtain the field value (which they have access to since they're in the same class as the fields) and then return them to you... the field value is protected.
  - The getter is a sort of middle-man that allows other classes to use field values without being able to manipulate them.
  - Always; even if you are making a public field for some reason, your code will be more secure if you configure it to be accessed by a getter.

When should you use getter methods?

How are they formatted?

What are setter methods?

How are they formatted?

What is setter validation?

```
data type method name
 (get + name of field, in camel case)
public double getLength() {
 return length;
 ^ the Field
}
```

- public method (in class body) that sets or updates the value of a field.
- if you want the user to be able to change the value of a field after initially declaring it, setter methods are the most secure way.

```
method name
(same convention as getters)
arbitrary name of
the value that the
user will enter; setter
takes this new val
as an argument
since there's
no return value
field name
data type
public void setLength (double length1) {
 this.length = length1;
 ^ setting a new value
 for the current object
 (in that field)
}
```

- If the user tries to set a field to an improper value (i.e. a String if the field type is double; a negative number for a length value, etc.)... there needs to be a way for the setter method to check & validate the incoming value.
- Add code to check for an illegal value, and if one is detected, throw an error to the setter to end the program.

## Example of validation?

```
public void setLength(double length) {
 if (length <= 0) {
 throw new IllegalArgumentException ("negative number");
 }
 this.length = length; }
 update the value last,
 only if validation passed
```

explanation of  
why its invalid

## What are derived fields?

- an imaginary "field" that is actually just a calculation or transformation of other fields.
- doesn't need to be stored in the class fields & doesn't need to be one of the parameters of the constructor
  - for example, the area of a square object when one of the class fields is "length"... You can derive area from the length value.
- Rather than storing them, write a getter method for it that includes the calculation
  - the calculation can just be done on demand inside the getter method.

## How do we store them?

```
public double getArea () {
 return length * length; }
```

## Example of a class with encapsulation topics covered so far?

```
Java file → public class Square {
private, encapsulated fields declared private double length;
private String color;

public Square (double length, String color) {
 this.length = length;
 this.color = color;
}

public double getLength () {
 return length; }
public String getColor () {
 return color; }
}
```

public constructor

public getter methods for  
encapsulated  
fields

## What are immutable objects?

- Values that cannot be changed after being initialized.
- if an object's fields are immutable, then so is the object itself
- This is a good thing to have. By making fields private and writing getter methods, we have made our object immutable.

## - Interfaces -

What are Interfaces?

- An abstract data type that serves to provide a well-understood description of every method that the class promises to provide.
- Similar to a class in that ;
  - it is defined in its own .java file
  - the interface name is a "type"  
(Just like with classes; The `Triangle.java` class defines a new object of type `Triangle`)
  - the name of the interface should be the same as the name of the file

How do they differ from classes?

- Classes : need to fully define the object... Fields, constructor, methods.
- Interfaces: hardly any code, JUST a list of method signatures ; no Fields

What goes in an interface?

- It is a list of methods (JUST their names,not the coding of them) that something that implements this Interface is promising to provide.
- The Interface as a whole (& therefore all of its methods) are declared publicly - they have to be, so that impl class can use them.
- Its sort of like a contract ; every class that implements a particular Interface must include a coded implementation of each method defined in the Interface, and only those methods.
- Classes implement Interfaces ... they specify this in the class definition using the "implements" modifier.
- implementing classes must declare the methods as public

What are the naming conventions for classes & interfaces?

- In an example of a program seeking to create an abstraction of just 1 object , there will only be one class implementing the interface.  
(although it is possible for several classes to implement 1 interface)
- Therefore , the Interface is typically named after the object type , and the class is "[object type] Impl"

① static methods

- i.e. , methods that are related to the abstraction, but not specific to any given object instance

- Because (unlike instance methods), they don't need a specific instance or any of its fields to be implemented.
- ② default methods (the ONLY time an instance method is allowed in the interface)

Why are static methods allowed?

What is the second thing?

What are default methods?

- Instance methods that can be implemented / coded entirely, using other methods of the interface
- As opposed to methods that access and work directly with the field values

return side1.length() + side2.length();

cannot be a default method

return get(x) - other.get(x) + get(y);

can be a default method

What do we do with them?

- b/c of their nature, they can actually be defined in the interface rather than a specific implementing class ... in the interface, we define the method with the keyword "default" at the beginning;

SomeInterface.java

default double someCalculation()

return get(x) - other.get(x) + get(y); }

What is special about default methods?

- Implementing classes can still choose to define their own implementation of the method!
- Or they can use the default... they have that choice.

### CLASS:

TriangleImpl.java

public class TriangleImpl implements Triangle {

private double ax;

private double ay;

public Triangle (double ax, double ay) {

this.ax = ax;

this.ay = ay;

}

public double dist (double ax) {

.. ..

return }

3

• class definition

] fields

] constructor

] coded implementation  
of methods  
declared by  
interface

### INTERFACE:

Triangle.java

interface definition

list of all  
method signatures

public interface Triangle {

double dist (ax);

...

}

How do you "program to the Interface"?

- When you create new instances of the abstraction object (like in the main method), create them as objects of type [Interface] ... rather than as "type [object name]"
- We want to store all the objects of all of the classes that implement a certain interface ... as objects of the "interface type"

Example of programming to the Interface?

```
public static void main (String [] args) {
 Triangle t1 = new Triangle Impl (3, 4);
}
```

name of Interface                              name of object class

(NOT)

~~Triangle Impl t1 = new TriangleImpl (3,4);~~

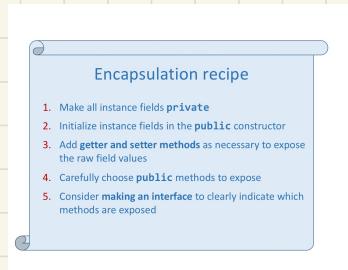
What is the advantage of encapsulation?

SUMMARY: What does encapsulation do?

SUMMARY: How do we support encapsulation?

Encapsulation "Recipe":

- Several classes can implement the same Interface.
- With encapsulation, we can create new implementations of the same behavior and use them in our programs -- and no one has to be the wiser.
- separate an abstraction into two parts :
  - ① Interfaces : publicly describing everything that the object/abstraction can do.
  - ② Classes : Implement the methods dictated by the Interfaces.
- Define abstractions as one or more interfaces:
  - \* getters and setters for direct and/or derived properties
  - \* other methods that are part of the abstraction.
- And write classes that implement one or more interfaces:
  - \* all fields within a class are marked as private.
  - \* public constructor
  - \* methods that implement any interface(s) must be public.
  - \* Internal methods marked as private.

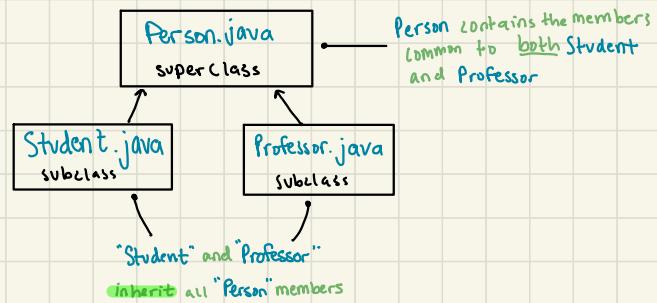


# Inheritance

What is inheritance?

→ 0.5 mm thickness  
For headlines... 0.15 for footnotes  
(this current size)

- Factoring common code into a superclass.
- When several classes in a program have several commonalities in their fields and/or methods (members), we can create a class that contains only the common members — a **superclass**.
- As well as the individual classes that use ("extend") that superclass — the **subclasses**.
- For example a **superclass** that contains "address," "name," and "age" fields & methods could be called "Person"
  - a subclass named "Student" could extend that class, as well as add its own fields (like "credit\_hours") that are unique to the subclass object.
  - can also add its own unique methods.



How do you declare superclasses and subclasses?

- superclasses: normal Java classes. Nothing different or special about them.

```
public class Person {
 private String name;
 public Person (String name) {
 this.name = name;
 }
 public String getName() {
 }
```

```
 return name;
}
```

- subclasses: declare the inheritance with the keyword "extends":

```
public class Student extends Person {
 ...
}
```

• even though the class body is empty, a "Student" is a "Person", so it automatically inherits all class members of the Person class.

• e.g., Student automatically has a `getName()` method and a `name` field.

- every time we create a new instance of the subclass (e.g. `Student`), we are also constructing a new instance of the superclass ... to construct a subclass object, we do not have to code a new constructor. Instead, call the superclass constructor using **Keyword `super(parameter args)`**
- the parameter arguments are the same parameters that the superclass constructor takes in.

How do you write the constructor of a subclass?

→ essentially the same concept as calling a class's method.

```
public class Student extends Person {
 public Student (String name2) {
 super(name2);
 }
}
```

(can name  
this  
whatever you want)

→ calling the `Person` class constructor, passing in "name2" as the String for the "name" parameter requested by the `Person` class constructor.

→ Technically, nothing! All of the fields & methods of the superclass automatically exist in the subclass

→ The returned reference for a new object of the subclass can be declared as either a subclass type or its superclass type — both references point to the same memory address on the heap!

```
Person jillFisher = new Professor("Dr. Fisher");
or
```

```
Professor jillFisher = new Professor("Dr. Fisher");
```

\* basically multiple "identities" for the same object — this is an example of **subtype polymorphism**.

→ When a file / class / interface etc. extends more than one "Super" class.

→ it is NOT allowed for classes — subclasses in Java can only have 1 parent class.

→ For interfaces! A **subinterface** is simply a union of all the methods declared in all of the parent interfaces.

→ Yes! An extended interface is one that adds methods to an existing interface

→ A class that implements an extended interface (aka "sub-interface") is required to provide methods for that interface AND its parent(s) (aka "super-interfaces")

→ When you want to define a new type which is a combination of existing interfaces, and need a single specific object that implements same combo of interfaces

\* want a single type/object to represent a specific combination

\* (REMEMBER: the whole point of any individual interface file is to define/be a contract for a class/object)

→ Often, inheritance is just used to pick methods from a few interfaces & pull them together into a new interface name — not even adding any new methods

\* the subinterface provides a typename for that specific combination

→ 2 existing interfaces:

① `public interface Tossable {`

```
 public void tossTo (PointInSpace target);
```

What else does the subclass need besides a constructor with "super"?

How do subclass objects exist in memory?

What is **multiple inheritance**?

When is multiple inheritance allowed?

So inheritance & class extensioning can also be applied to **Interfaces**?

When would you need inheritance for Interfaces?

Example of mult. inheritance with **Interfaces**?

"this interface says that "if you are an object that is tossable, you can toss it to some target in space

(2) public interface Trackable {

PointInSpace getPosition();  
Vector getVelocity();  
Vector getAcceleration();

"if you are an object that is trackable, I can get your position."

What type of object will we provide to the "juggle" method?

→ new function we want to create : "juggle" function that takes in 3 objects that it is going to juggle

→ 3 "objects that are trackable"? (aka place method in Trackable interface)

NO!

• How do we know whether we can toss the objects in the first place? Can't juggle without tossing.

→ 3 "things that are tossable"? (aka place juggle in Tossable interface)

NO

• If the objects tossable but not necessarily trackable, then how will we track the objects in order to catch them after tossing them?

→ Solution: create a new interface that extends Tossable AND Trackable

public interface Jugglable extends Tossable, Trackable {  
 }

• even if we have no new method to add to this interface, being "jugglable" is just the combination of being both tossable & trackable

• defining this new interface is the only way to define this new juggle function, because juggle requires 3 objects that implement both tossable & trackable

→ Since we gave the 'tossable & trackable' combo a new name, we can create classes (objects) that are an impl of Jugglable - both tossable & trackable - and can then be fed into the juggle method without error.

↳ static void juggle (Jugglable obj1, Jugglable obj2, Jugglable obj3);

→ So multiple inheritance with Interfaces is essentially a workaround to the single-inheritance rule of classes ...

• a class for a "redBall" object, for example, cannot extend both Tossable & Trackable because multi-inh isn't allowed. HOWEVER, it can implement an interface which extends both of those... essentially creating the same effect.

So what is the point of making this new interface, "Jugglable"?

## Polymorphism - "many forms"

- the principle of reusing one common name or symbol to refer to many different related things.
- big concept in OO-programming; shows up in many different ways.

What are some examples of Polymorphism?

### ① Type Polymorphism

- When an interface has multiple implementation classes
  - e.g., *Nigiri* and *Sashimi* are 2 classes that both implement the *Sushi* interface.
- When a class has multiple subclasses
  - e.g., *Student* and *Professor* are both "Person" objects as well as *Student/Professor* objects.
- using generics ( $<T>$ ) so a Field or variable can take on different data types.
  - e.g., *ArrayLists*; *ArrayLists* can be of any type; the type is declared by the coder at the time that they are using it.

### ② Parametric Polymorphism

- several different implementations ('versions') of the same method
- There are 2 versions of this:
  1. method Overriding - inherited method is overridden & replaced in the subclass.
  2. method Overloading - 2 methods with the same name but different in the parameters that they take.

### ③ When multiple methods have the same name

- multiple diff. versions of the constructor
- constructor overloading - constructors with difference in parameters/arguments
- allows programmers to program to a specific subset of an object's members.
- allows programmers to group similar (but different) entities or behaviors together and program to their common type.

### - Type Polymorphism -

What are "is-a" relationships?

- Every (subclass type) is-a (superclass type) but not every (superclass type) is-a (subclass type)
- Every *Student* is-a *Person* → Every *Professor* is-a *Person*
- Not every *Person* is-a *Student*

What is the "instanceof" operator?

- A Java operator used to test is-a relationships
- Ex: *Person jane = new Person ("Jane");* } defined Jane as a person
 

```
if (jane instanceof Professor) {
 System.out.println("Jane is a professor");
}
```
- This will come out false... hasn't specified what 'type' of person Jane is.

What is type-casting?

Example of type casting?

- Turning an object into a different kind of object (from one 'type' into a diff 'type')
- An object can only be typecast to another object if there is a guaranteed is-a relationship (i.e. Student is-a Person)

Person Kmp1 = new Professor ("Kmp");

- creating a "Professor" object, but the reference/type is a "Person"

- Kmp1 can only access the "person" part of the created object, because of the type that we have associated with the name.

- Now, we want to create a new object that typecasts "Kmp1" into a Professor type rather than a Person type;

Person Kmp1 = new Professor ("Kmp");

Professor Kmp2 = (Professor) Kmp1;

use parentheses with the desired type inside in order to type cast the particular object to that type.

initially

- a type of typecasting where you take a reference to an object that is typed as the parent class, and force it to be one of the subclasses.

→ basically what we did in the example above... turning it into a more specific type.

→ the opposite.

- Since we are going from subclass → class, we know that it is ALWAYS going to work. Therefore, we actually don't need to perform the typecast; the compiler can assume the casting process.

→ i.e., upcasts are usually implicit; we don't have to do the parentheses thing; compiler can implicitly do it (like w/ syntax errors)

- At compile time. (aka a red line error that is shown before you run the program) compiler can definitively confirm (or deny) the upcast using the declared type relationships in the code.

because they can be checked, explicit casting is unnecessary

- At runtime (aka after you hit play & run the program)... the computer will not tell you if your downcast is invalid until you run the program.

→ If the downcast is not valid, the computer throws a ClassCastException.

Person Kmp = new Professor ("Kmp");

← A person object, of type Professor

Student Kmp3 = (Student) Kmp; ← ERROR... cannot typecast a professor into a student

- Every Impl class automatically has an is-a relationship with the Interface it is implementing; PositionImpl is-a Position, for ex.

→ Yes! A subclass extends its superclass & therefore inherently also extends the superclass' Interface(s). Think of "implements" as being at the top of the inheritance hierarchy.

How does the computer 'check' upcast attempts?

How does it check 'downcast' attempts?

Example?

How do is-a relationships work with Interfaces?

Can inheritance & implementation exist together?

Example of this?

class A implements InterA { }

- "A is-a InterA"

class B extends A implements InterB { }  
• "B is-a A"

- "B is-a InterA"
- "B is-a InterB"

class C extends B implements InterC { }

- "C is-a B", so "C is-a A" too
- "C is-a InterA" • "C is-a InterB"
- "C is-a InterC"

How can we figure out the hierarchy of an entanglement like this?



→ We can reason about these types as a graph system;

→ From this graph, we clearly know which classes have relationships with each other, which will help us figure out which **upcasting** is & isn't possible.

- **RULE#1:** An object can only be upcast to another if the 2 have a **is-a relationship** -- can I reach object X from object Y by following a forward path of arrows? If not, no guaranteed **upcasting**.

Upcasting in this example?

B test1 = new B();  
InterA testAsIA = (InterA) test1;

"test1" is upcast  
from a B object to  
an InterA object

→ We know that "B" & "InterA" have an **is-a relationship** (chain of arrows)

B test1 = new B();  
InterC testAsIC = (InterC) test1;

**COMPILE-TIME**

**ERROR:** we can't definitively say that  
all B objs are going to also be InterC objects... they don't have an **is-a relationship**.

- Method Access -

*(an object that supports several types,  
either due to inheritance or implementation)*

What methods can a **polymorphic** object access?  
Ex?

→ When calling a method to an object, you only have access to the methods defined for the type that you declared the object as (i.e. its reference).

→ Student avik = new Student("avi Kumar");

• avik object has access to all methods in the Student subclass, like "int getCredits()", "String getStatus()"

• Since Student.java extends Person.java, it also has access to Person's methods.

Person avik2 = new Student("avi Kumar");

**VERSUS**

Student avik2 = (Student) avik;

• even though avik is a student, it has been declared as a Person type & thus only has access to the Person.java methods (like getName())

upcasting example

What is the "Object" class?

- The Mother of all classes ... parent class to all classes in Java.
- All classes automatically & implicitly inherit (& extend) the Object class.
- Any object of any class can be instantiated as an object of type Object (polymorphism)

`Object avik = new Student ("avi kumar");`

\*However, they are then restricted to only the methods defined in the Object class.

- `public boolean equals (Object o)`
- `public String toString()`

What methods are defined in Object class?

# Overriding and Overloading

Recall the 3rd type of Polymorphism  
(4 pages ago)

- ③ When multiple methods have the same name
  - several different implementations ('versions') of the same method.
  - 2 versions of this: Overriding & Overloading
- When you inherit a method from a parent class, and replace it.
  - Since subclass has already inherited that method, overriding isn't necessary; without it, the method (as defined by parent class) will still run.
  - Subclass desires to have its own subclass-specific implementation of the method.
  - Providing multiple versions of the same method, but which differ in some way.
    - in particular, they have to differ in their parameters (the arguments they take in).
    - can also overload constructors.

## - Overriding -

When is an overridden method going to be called?

- Anytime it's called in relation to its corresponding object — even if the reference is as a parent class.
- e.g., say Professor has overridden the Person getName() method to add "Dr." in front of the getName() string. both of the following objects will utilize the overridden method:

```
Professor emily = new Professor("emily");
Person emily = new Professor("emily");
```

- a one-word token preceded by the "@" sign that is used to hint to the compiler the role or constraint of whatever is coming next
- Compiler can check to make sure the constraint is true & works.
- Compiler directives are not necessary, compiler will still work & compile code without it.
- useful as a backstop to help you check for typos & small bugs & etc. You're basically telling the compiler "make sure I do what I'm telling you I'm going to do, and if not then warn me."

Why use compiler directives?

- a compiler directive that will make sure that the method in question is actually a method you've inherited
- goes on its own line directly before the overridden method.
- They are OPTIONAL

Can we access parent class fields when rewriting methods?

→ NO! When writing an overridden method in the subclass, you run into the problem that the parent class' fields are private, so we can't use them;

```
Professor.java @Override
public String getName() {
 return "Dr." + name;
```

\* this will NOT work because "name" is a private field in the Person superclass.

How do we resolve this issue?

## → 2 Solutions :

- 1) change the access modifier for the parent class field(s) from `private` (member only accessible inside class body) to `protected` (member accessible only from inside class & subclasses).  
" `protected String name;`"

→ not the best solution because we have relaxed the protection of a field for ALL subclasses just so that it can be accessed by 1 subclass.

→ goes against encapsulation

- 2) call the original (not overridden) method from the parent class using the `super`

Keyword

`Professor.java`

```
@Override
public String getName() {
 return "Dr." + super.getName(); }
```

calling parent class' `getName` method, which DOES have access to the private fields.

→ "super" restricts you to the superclass version of the method.

→ The idea of always going to the most-overridden version of a method (applicable to the object), regardless of the assigned type.

→ In Java, all methods are virtual. (But not other languages)

→ the `super` keyword is kind of like a toggle / "escape" if you want to temporarily turn the virtualness off. "Virtualness" is the default way.

→ makes fields/methods/objects immutable

What is a `virtual` method?

→ "super" restricts you to the superclass version of the method.

→ The idea of always going to the most-overridden version of a method (applicable to the object), regardless of the assigned type.

→ In Java, all methods are virtual. (But not other languages)

→ the `super` keyword is kind of like a toggle / "escape" if you want to temporarily turn the virtualness off. "Virtualness" is the default way.

→ makes fields/methods/objects immutable

What is the `final` keyword?

→ it means that that method cannot be overridden by a subclass

→ makes overriding "illegal."

```
public final String getName() {
 return ... ; }
```

→ means that the value of the field/variable can never be changed after instantiation.

`final String name;`

→ means that that class cannot be extended or have subclasses.

```
final class Professor {
 }
```

What does it mean when `final` is added to...

a method?

→ it means that that method cannot be overridden by a subclass

→ makes overriding "illegal."

```
public final String getName() {
 return ... ; }
```

→ means that the value of the field/variable can never be changed after instantiation.

`final String name;`

→ means that that class cannot be extended or have subclasses.

```
final class Professor {
 }
```

a field or variable?

a class?

## - Overloading Methods -

How do we distinguish between  
2 overloaded methods?

- Even though they have the same name, the parameter lists must be different
- for ex, one `public void promote()` & another `public void promote(int status)`
- the return types do not have to be the same.
- Another ex: a method that takes 5 parameters, and another method of the same name that takes 2 parameters &, within the method, sets default values for (what would have been) the other 3 parameters.
- They have to have the same access modifier
- Have to have the same static/non-static status.

Requirements of overloaded methods?

## - Overloading Constructors -

- providing multiple constructors; they have to have different parameters
  - so that the compiler knows which constructor you are calling (based on the arguments that you put in.)
- Ex: • A professor constructor that takes in `String name` & sets default status to 0.  
• AND a professor constructor that takes in both a `String name` & an `int status`
- It's convenient! Allows you to make context specific versions of a constructor (or method) to perform the same action in different situations.
- It is rarely necessary — just makes coding easier.
- the way for one constructor to call a different overloaded constructor.
  - e.g. if you want to create a second constructor that is just a 'special case' of the first generalized constructor
  - this applies to the Professor example above
- To call a different overloaded constructor, the 1st line in your current constructor should be `This ([parameters of constructor being called])`
- If a constructor is already using `This()`, it doesn't need the super() call.

Why use multiple constructors?

What does it mean to chain  
constructors?

How do you do constructor chaining?

Example of chaining  
constructors?

```

constructor 1 [public Professor (String name,int status) {
 super(name); • calling parent class.
 this.status = status; • setting object's status field.
}

constructor 2 [public Professor (String name) {
 this(name, 2); • calling constructor 1
}

```

\*essentially, constructor 2 creates a new Professor object with the status pre-set to 2. \*

# Parametric Polymorphism

What is a generic type?

- Using generics ( $<T>$ ) so a field or variable can take on different data types.
- A class or interface that takes a data type as a parameter.
- For example, Lists and ArrayLists -- they have to be declared with a specific type;  
`ArrayList<Student> studentList = new ArrayList<>();`  
↳ declared type

How is a generic class defined?

- After the name of the class, use  $<>$  with a list of type parameters inside, to act as placeholders for real data types.
- Basically, we don't know the type that the class will use until it is used. So instead, we create the class using an arbitrary placeholder data type -- the type parameter.
- the type parameter (TP) isn't an actual data type
- We define all the code in the class w.r.t. the TP, acting as if it is a real data type ... and THEN, when the class is called, the user declares the actual data type that they want to use.  
↳ all of the mentions of the TP inside of the class get replaced with the actual data type name.

Container.java

```
public class Container<TP> {
 private TP contents;
 public Container(TP item2) {
 contents = item2;
 }
 public TP getContents() {
 return contents;
 }
}
```

Main.java

```
Container<String> food = new
 Container<>("fries");
Container<double> price = new Container
<>(1.67);
```

What types of data can be used in generic type classes?

- Once an object has been created with a declared instance type, the type can't be changed.
- ONLY reference types -- a.k.a. only objects (no value types).
- `Container<int> numbers` would NOT work.

Then how do we use generic classes when working with value types?

- Java provides a corresponding reference type version for every value type.
  - for value type `int`, there also exists an `Integer` object ... object = reference type.
  - `bool` → `Boolean`      `char` → `Character`      and etc.
- If we want a `Container` with integers inside, we can use the `Integer` object class instead.  
`Container<Integer> numbers = new Container<>(2);`

How do we convert between  
Integers and ints (for ex.) ?

→ The compiler automatically does this for us. What that means is we can retrieve int values from a generic class of Integers w/o having to do an extra conversion step. Vice versa for adding int vals to a generic class of Integer vals

(Ex) `int retrieveI = numbers.get(contents());`

- would return 2 (see previous page)

## Reference types v.s. Value types

What is a value type?

→ A type that is defined entirely by its value — a string of ones and zeroes that is stored directly in memory in their specified location.

→ (Ex) int : stored as a string of ones & zeroes in 4 bytes of memory (remember int is size 4-byte)

→ byte short int long  
float double char boolean

→ everything else outside of these 8 items is an Object

→ They exist/are stored in the heap (not the memory)

→ We refer to them through a reference in the memory that points to the actual object in the heap

→ For all objects, the value of the variable (the declared name of an instance of the obj) is actually a memory address that is stored in the memory.

→ This memory address is a reference to the object — not the actual object itself. The mem. address points to the location in the heap where the actual object lives.

int num = 2 ;  
\*

\* this is a value type variable; the value of "num" is the actual string of 0s and 1s that comprise the integer (located in memory).

String name = "ella";  
\*  
Student avi = new Student("avi");  
\*

\* this is a reference type variable; the values of "name" and "avi" are memory addresses (located in memory) that then point to the actual String and Student objects (located in heap).

## Switch/case statements

What are they?

- basically just a fancy version of if and else-if statements.
- we take a value, go to each case statement inside the body of our switch & compare the value to the value associated with that case.
- The first time we find a match is when we start executing, & then we keep executing until either
  - ② we hit a "break;" statement or ③ we get through the whole body of the switch.
- The most commonly used format is where all of the cases are separate & each ends in a break statement to prevent you from falling into the next case.
  - \* However you can use them however you want; cases can fall into other cases.

How do you code a switch/case statement?

→ (Ex) Compare to an if-else statement:

```
public static String seasonToStr(int season) {
 String answer = "Unknown";

 if (season == 0) {
 answer = "Spring";
 } else if (month == 1) {
 answer = "Summer";
 } else if (month == 2) {
 answer = "Autumn";
 } else if (month == 3) {
 answer = "Winter";
 }

 return answer;
}
```

(*"season" is an int variable*)

```
switch (season) {
 case 0 :
 answer = "Spring";
 break;
 case 1 :
 answer = "Summer";
 break;
 case 2 :
 answer = "Autumn";
 break;
 case 3 :
 answer = "Winter";
 break;
 default:
 answer = "still unknown";
}
```

- the value that we are going to compare with each case.
- the value associated with the case, which gets compared with the main value.
- the action(s) that get executed IF the case val matches the main val.
- optional "default" case, if no other case provided a match but we still want to perform an action.
- the actions executed in default case.

# enumerations

- A programmer-defined data type that has a predefined, finite set of possible values.
- Enumerations - keyword `enums` - are usually declared in their own .java file OR directly in the body of a class

When are enumerations useful?

What is the value of these "possibility" objects?

Can we code without enumerations?

Example?

- When we want to limit the values that we are working with to a finite set of possibilities.
- The data type in an enumeration is essentially a list of the defined possibilities
- They don't actually have any inherent value associated with them - they are just symbols that help us write case-by-case code.
- Yes! They just make programming easier by making the compiler do the hard work.
- Refer to switch case example from previous page (seasons)
  - We are setting a String to a certain value based on what season it is, and we chose integers to represent the different 'season' possibilities... but it is up to us, as the programmer, to keep track of which integer corresponds to which season.
    - e.g. 0 = spring, 1 = summer, 2 = autumn, 3 = winter
  - Also, we have to worry about dealing with invalid or out-of-range inputs & other edge cases (which means throwing exceptions, writing more code, etc.)

How would an enumeration make this process easier?

- It does this job of converting an integer to a specific meaning for us!
- In an enum, we just create the set of symbols (compiler associates integers with the values under-the-hood, we don't have to worry about that)
- Now we have a specific type (e.g. Season) that can be used for a variable, & this variable will be restricted to only being one of the defined symbols (as opposed to an infinite number of integers), & can't be set to something nonsensical.
- Now we also get all of the nice type-safety & value-safety properties that come with using a type!
- One option is to make a separate class, except instead of "public class" you say "public enum";

```
public enum Season {
 SPRING,
 WINTER,
 FALL,
 SUMMER
}
```

- The declared name of the type
- The "Season" enum defines 4 possible values
- The limited list of all possible "Season" values
- These 'objects' serve as nothing more than symbols; thus, we don't need to define them further!
- Should be listed in all caps

How do you instantiate the enum object in another class?

- via the format `variableType. POSSIBILITYNAME;`
- for (ex) `Season currently = Season.AUTUMN;`

What is a real example of how using enums is easier?

→ COMPARE: Season case example using case statements - NO ENUM:

```
public static String seasonToStr (int season) {
 String answer = "Unknown";

 switch (season) {
 case 0:
 answer = "Spring";
 break;

 case 1:
 answer = "Summer";
 break;
 case 2:
 answer = "Autumn";
 break;
 case 3:
 answer = "Winter";
 break;
 default:
 answer = "still unknown";
 }
}
```

- season is an int, so (basically) infinite possible input values
- manually keeping track of which int corresponds to which season

→ VERSUS making the same statements with an enum:

```
public static String seasonToStr (Season season) {
 String answer = "Unknown";

 switch (season) {
 case SPRING:
 answer = "Spring";
 break;

 case SUMMER:
 answer = "Summer";
 break;
 case AUTUMN:
 answer = "Autumn";
 break;
 case WINTER:
 answer = "Winter";
 break;
 default:
 answer = "still unknown";
 }
}
```

- The method takes a Season enum input from user, rather than an int
- The job of having to assign int values is eliminated; just check each case for the value of the "Season" object.

# Composition and Aggregation

What is 'composition & aggregation'?

- "making smaller objects work together"
- 2 endpoints on a spectrum of program designs about how objects relate to the objects that they encapsulate
- Agg. & comp. are both relationships where one object encapsulates instances of other objects.
- The difference falls in the relationships between the outer and inner objects.
- Kind of like the ~complexity~ of a class
- For more complex programs, we build objects out of other objects rather than having just 1 outer class where everything is flattened down to just primitive fields - that is long & tedious
- to manage complexity & be able to reason about the relationships between objects in a more manageable way.

Why use this method?

How can we distinguish between simple & complex classes?

- Simple Classes
- fields are primitive data types ; int, string etc.
- a class is simply a "container" for its data
- Classes define operations on their fields

## Complex Classes

- encapsulated fields are objects themselves - not limited to just int, double, String, etc.

What is the difference between aggregation & composition?

- Aggregation
- the internal objects can exist independently without an outer containing object
- internal objects have (or have potential to have) a meaningful purpose & use outside of this object

## Composition

- The inner objects cannot exist without an outer containing object ; they aren't meaningful without it.

## - Aggregation -

What are some signs that a class is using aggregation?

- The encapsulated objects are provided externally.
- Some (or all) of the constructor's parameters are objects from another class
- Ex: Public Roll (Ingredient portion[] ing, String name) { ... }
- an entirely separately defined class
- There may also be getters & setters or methods w/ the ability to remove that object from the class.
- Encapsulated objects are also independently referenced outside of the aggregation.
- i.e., they have their own "lives" & utilities outside of the current class... including potentially being part of a different aggregation!
- Aggregation = taking independent things & giving them together.

## - Composition -

What are some signs that a class is using composition?

- Encapsulated objects are created internally;
    - usually within the constructor — the parameters of the constructor are usual data types, and ~objects~ are created from these parameters inside of the constructor, sort of 'on-the-spot'
    - usually no setters or getters for these internal objects — they aren't meant to be exposed to the outside world. **Exception — dependency injection.**
    - telltale sign: constructor usually doesn't take parameters
  - Encapsulated objects don't make sense outside of the abstraction
    - they usually aren't shared with other abstractions.
  - Encapsulated objects' functions & states are only accessible through the composition.
    - Only the current class can call the objects' methods or retrieve info (getters) about the object.
- Composition = having internal parts & organs that only belong to it.

## - Composition Over Inheritance -

What are the 3 approaches for writing a class that implements multiple Interfaces?

Option 1: No hierarchy

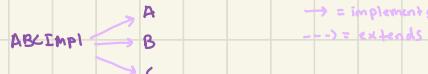
How does this look in memory?

→ Example for notes: A class "ABCImpl" which implements Interfaces A, B, AND C

- ① No hierarchy
- ② Inheritance
- ③ Composition

→ Public class ABCImpl Implements A, B, C { ... }

→ Directly implement all if the methods for all of the interfaces



→ Every instance of `ABCImpl` exists directly in the heap.

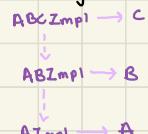
→ Works best if `A`, `B`, and `C` already naturally lend themselves toward some hierarchy

→ **Ex**

public class (pc) AImpl implements A { ... }

pc ABImpl extends AImpl implements B { ... }

pc ABCImpl extends ABImpl implements C { ... }



→ If `A`, `B`, `C` already related in some way, then this is a good approach to use.

→ Every instance of `ABCImpl` exists in memory as an `AImpl`, `ABImpl`, AND `ABCImpl` object (polymorphism)

How does this look in memory?

### Option 3: Composition

- A, B, and C each have their own basic implementation objects (AImpl BImpl CImpl)
- Then, ABCImpl will implement A, B, and C ... but rather than explicitly defining each of the methods dictated by the interfaces, it will designate its own internal private A object, B object, and C object
- So whenever ABCImpl needs to do something that it promised to do as an impl of A/B/C, it just turns around and uses its internal A/B/C object to do that thing.
- Approach works best if A, B, and C are separable/independent (unlike with Approach 2)
- `public class ABCImpl implements A, B, C {`
- `private A aObject;`
- `private B bObject;`
- `private C cObject;`
- `...`

How does this look in memory?

Bottom Line: when do I use inheritance versus composition?

- When the problem seems hierarchical: Inheritance
- When the problem components are independent: Composition
- When given the choice... FAVOR COMPOSITION!

## Abstract and Concrete Classes

What is an abstract class?

- A class marked "abstract" cannot be constructed directly, because it contains abstract methods.
- If there is at least one abstract method in a class, the whole class must be marked as abstract.
- Must have subclasses; an obj of the abstract parent class cannot be directly instantiated... has to be a specific "type" (subclass) of itself.
- A method that is defined in the parent class (so that any Parent-Class object can access it)... BUT it has no implementation (method body) that defines what it does. The method needs to be overridden & specifically defined by every subclass of the parent.
  - As opposed a concrete parent class, where every method is implemented, but subclasses can still choose to override them.

Why would you opt to use an abstract method?

- If there is no sensible implementation at the parent class level, BUT it does make sense for every instance of the superclass to have access to the method. (e.g. Person avi = new Student("avi"); is still a Person object.)
- For ex a Person parent class has Student, Professor, Adjunct, Researcher, & Counselor subclasses. A getStatus() method makes sense for every subclass (eg the position that they hold as a student or employee), but it doesn't make any sense for the parent class (a "Person" doesn't have a status, yk).
  - Therefore, we mark getStatus() as abstract within the parent class so that we don't have to define it there... & each subclass is then required to define its own implementation.

OK so why even use a parent class/inheritance at all, if it has to be abstract?

- Remember, parent classes are defined as abstract even if just 1 method is abstract... There can still be other regular methods that make subclassing useful.
- The purpose of the parent class is to gather all common properties in 1 place for convenience; It defines everything that an object of that class should be (including all objects of its subclasses).
- Classes that can be directly constructed because nothing is missing.

What is a concrete class?

```
public abstract class Person {
 private int status;
 public Person(String name) { ... }
 public String getName() { ... }
 public abstract String getStatus();
```

• effectively declares Person as an abstract class, meaning that plain Person objects can NOT be constructed; Person jane = new Person("jane"); -ERROR -

• Unlike the other methods, this abstract one has no curly brackets with coded definition.

Can a class be marked abstract even if there are no abstract methods?

- Yes! Sometimes we may want to mark a class as abstract even if it is fully defined.

Why mark a class "abstract" if it is fully defined?

- If we want to forbid the creation of an object at parent class level, and force it to use a subclass (even though all the parent class methods have reasonable implementations for subclasses to inherit)
- Basically depends on your(the programmer's) intentions with the project/abstraction.

# Dependency Injection

What is coupling?

→ When the definition of one class has a line of code that references a different class by name, it creates & enforces a dependence between the 2 classes.

• For ex: `public VehicleImpl (int radius, String name) {`

`WheelImpl frontLeft = new WheelImpl (radius);`

`WheelImpl wheel2 = new WheelImpl (radius * 2);`

`}`

• By calling for a new object of the `WheelImpl` class everytime a `VehicleImpl` object is constructed... these 2 classes have formed a dependency.

→ Classes which reference each other by name cannot be used independently.

→ The more dependent that a project's classes are with each other, the harder they are to separate - but this isn't ALWAYS a bad thing. Sometimes classes don't need to be separated...

→ Between classes which are in the same package, because they are always going to exist together

- recall the Java imports that you sometimes add to the top of a .java file (like `Math`, `ArrayList`, `HashMap`, `Scanner`, etc...) those are all packages of multiple class files being imported together.

→ When we build larger systems that use multiple packages & packages start to depend on each other.

→ The best way to do cross-package couplings is at the level of the Interface so that we don't make assumptions about implementation.

• i.e. having `private Wheel frontLeft;` as a private field in `VehicleImpl.java` isn't problematic on its own.

## Highly Coupled Code

• many named references between class files, even if they aren't closely related.

## Loosely Coupled Code

• separated into well-defined independent modules

→ Systems that are loosely coupled are easier to maintain, modify, read, & etc.

```
→ public class VehicleImpl {
 private Engine engine;
 private Wheel frontLeft;
 private Wheel frontRight;
 private Wheel rearLeft;
 private Wheel rearRight;
```

RECALL: referencing the Interfaces = NOT problematic

```
 public VehicleImpl(){
 engine = new EngineImpl();
 frontLeft = new WheelImpl();
 frontRight = new WheelImpl();
 rearLeft = new WheelImpl();
 rearRight = new WheelImpl();
```

→ Injecting `Wheel` and `Engine` objects into the class through the constructor

→ `VehicleImpl` is tightly coupled with a specific implementation of the

Engine & Wheel Interfaces

• we can't make different choices about what kind of wheel or kind of engine (for ex, if we wanted to use some subclass of `Wheel`)

Example of a highly coupled class?

How do we design a composition that supports low-coupling?

→ Dependency Injection!

→ Writing a composed class in such a way that allows us to inject which specific instance of another class object to use... rather than the comp. class hard-coding this into the constructor.

→ An object/class receives other objects that it requires/depends on, as opposed to creating them internally.

→ There are several ways to support DI.

What is Dependency Injection?

What is one way to support DI?

→ Inject the other classes' objects into the composed class through setter methods

```
public class VehicleImpl {
 private Engine engine;
 private Wheel frontLeft;
 private Wheel frontRight;
 private Wheel rearLeft;
 private Wheel rearRight;

 public VehicleImpl() {
 engine = new EngineImpl();
 frontLeft = new WheelImpl();
 frontRight = new WheelImpl();
 rearLeft = new WheelImpl();
 rearRight = new WheelImpl();
 }
}
```

Replace this... with this:

```
public void setEngine (Engine e) {
 this.engine = e; }

public void setFrontLeft (Wheel f1) {
 this.frontleft = f1; }

... and so on
```

What is another way to support DI?

→ This is known as **setter injection**.

→ **constructor injection**.

→ the dependencies are given to the class at the time of construction — as parameters

• Replace this... with this :

```
public VehicleImpl (EngineImpl e, WheelImpl f1, WheelImpl fr, WheelImpl bl, WheelImpl br) {
 ... (constructor code)... }
```

→ You can also incorporate both setter & constructor injection.

→ The class is still considered a composition in terms of object design ; the objects being injected into the class are still its "internal components" —

- They are specific to this instance of the class & not used in other instances of the class (e.g. a distinct, different `WheelImpl` object goes into each instance of `VehicleImpl`)
- They don't make sense outside of the larger concept of the composed class.

→ Btw by supporting DI, it does start to look more like an aggregation

DI in Composition

- have to make an active choice (and write diff code) to support DI
- sort of goes against a composed class' nature; exposing our 'internal organs' slightly more than we would want to.

DI in aggregation

- aggregations already basically support DI by design & definition... a built-in feature of an aggregated abstraction.

How does DI change a class' status as a 'composition'?

What is the difference between DI in compositions versus aggregations?

Bottom Line: What are the pros and cons of dependency injection?

Advantages

- Makes objects more configurable — more true to the idea of "programming to the interface"
- Easier to write isolated unit tests
- promotes loose coupling of classes

Disadvantages

- Requires more code to construct a new object (like in the `Main.java` class), and user needs to know about all of those dependencies being injected.
- Requires more development effort
- Goes against "convention over configuration"

## Inversion of Control

What is "traditional" control flow?

- We start execution in the `main()` method, & we write everything else straight after — aka just the sensible intuitive "flow" of writing a program.
- Since we wrote the first procedure on the stack frame, we have full control over the execution of the program

What is inversion of control? (IoC)

- Sometimes a program is split up into 2 parts (perhaps 2 pieces written by 2 independent, separate developers) ... 1 of these programs
- Basically, imagine we have developed a piece of software that still has some missing components / can be made more specific by a missing ingredient
- And now imagine there is another separate piece of software (for ex, something created by a different developer) which contains these 'missing components'
- When we are executing the flow of our program, there are key moments where we need to rely on the other software's components in order to fill in the missing answers in our algorithm ... when we call that other component, we are effectively ceding control to that program
- And vice-versa; if my program is that other component, I'm in a situation where I don't get to control the execution flow:
  - Someone else decides when to call my methods & I am only allowed to respond to those method calls
  - Execution jumps in & out of my methods
- This relationship between the 2 components is called Inversion of control.
  - basically, we as the programmer don't always get to control top-level execution.
- IoC is a common programming pattern for frameworks.
  - For ex, think of a user interface with buttons, & the code/software that performs the actions after the button gets clicked.
- DI is considered a form of IoC
- DI is a useful paradigm for programs that use IoC

What does Dependency Injection have to do with IoC?

Example of IoC?

- Array sorting with the Java collections framework
  - `ArrayList` has a sorting method which takes a 'comparator' interface as a parameter

```
ArrayList<int> list = new ArrayList<int>();
list.sort(comparator c);
```
  - the Comparator interface contains a method to order the objects (but since it's an interface we can override the method to create any sort that we want - like merge sort, bubble sort etc.)
  - the `sort()` method is written entirely w/o any knowledge about what the sorting algorithm will look like - `sort()` doesn't care about that, its only goal is to present a sorted list of an `ArrayList` object and provide generic code to do that.

- But when `sort()` is actually called, at some point there is that critical moment where it needs to retrieve the elements in sorted order — at that point, the `comparator` object is injected into that method and control is 'inverted' to the `comparator`.
- The relationship between `sort()` and `comparator` is an example of IoC.

## Are private members inherited by a subclass?

Yes!

What does the internet (Oracle) define "inheritance" as?

How do we define "inheritance" in this course?

- to mean whether or not the private fields & methods of a parent class are accessible by the code in a subclass
- Therefore, Oracle says that private members aren't inherited by a subclass
  - However, in this class, we regard this conclusion as inaccurate
- to mean whether or not the private fields & methods of a parent class are part of the subclass object created in memory
  - Therefore, we answer the given question as YES !

# Notes: Midterm Review Session

= terminology

= key point

## - Git Stuff -

What is build automation?

- The process of converting your source material into a product that you can actually give to consumers — creates a shippable software product
- The 'build process' is what does this.
- They automate the entire build process for you
- Our build automation tool: **Maven**

What do build automation tools do?

What is...

a POM file?

- The main project file that you use in a Maven project; the configuration file
- written in XML & is typically placed in the root of the project folder.
- specifies various settings for the project
- Any package or 3rd party library that your project uses
- The template used for creating a new (Maven) project.
- The final end-product that Maven gives you
- e.g. the packaged output file that is produced at the end by a project.
- A configurable build process task; All of the tasks that Maven is automating (i.e. the "build tasks")

a Dependency?

an Archetype?

an Artifact?

the Life Cycle?

What does `git checkout` do?

- A command typed in the terminal that changes which branch we are currently working on (e.g. the branch that will move forward with each commit)

→ `git checkout [name of desired branch]`

review ppt  
says to use  
"git status" but idk that's right??

1) `git add .` (adds all files to new commit) OR `git add "[filename]" "[filename]"` (only add specific files)

2) `git push origin master` (send code to version control)

(only add specific files)

3) `git commit -m "[note provided for commit]"`

2) `git pull` (gets most recent updates from version control)

) `git merge` (merges those changes with our local changes)

(only use if changes are conflicting)

## - Access Modifiers -

What is `private`?

- Keywords that control the visibility & accessibility of methods, variables, & fields in a class.

What is `default`?

→ member only accessible inside class body

→ member accessible from anywhere inside the package

• packages example: all of the files & code in our assignments so far has been in one specific package.

What is `protected`?

→ member can only be accessed within parent class & its subclasses.

What is `public`?

→ member can be accessed by anyone

## - Anatomy of a Class -

```
public class Point {
 private int x;
 private int y;

 private static final double EPSILON = 0.001;

 public Point(int x, int y) {
 this.x = x;
 this.y = y;
 }

 public double distanceTo(Point other) {
 return Point.distance(this, other);
 }

 public static double distance(Point a, Point b) {
 return Math.sqrt(Math.pow(a.y - b.y, 2) + Math.pow(a.x - b.x, 2));
 }
}
```

What are...

instance fields?

class fields?

constructor?

instance methods?

Class methods?

What are "class members"?

- the variables which define the attributes of the class
- the static versions of these fields
- How you create & initialize an instance of the class
- the functions used to fill the fields or perform actions related to each instance of the class.
- Static version of instance methods - defines a method; isn't associated with any particular instance
- The methods & fields of a class. But NOT the constructor.
  - have to be defined by the **static** keyword... otherwise it's an instance field/method (member)

## - Encapsulation Principles -

What is encapsulation?

What is the 1<sup>st</sup> principle of encapsulation?

2<sup>nd</sup> principle?

How do we support encapsulation?

- pillar of OOP
- combining data (i.e. attributes & fields) and methods all together into one class ("binding data with the operations performed on that data.")
- Used for hiding the representation of an object from anywhere outside the class.
- Do not expose the internal state of an object directly - eg declare private fields
  - protects instance fields from being accidentally changed
  - Allows internal code to be refactored w/o breaking external code.
  - essentially separating what you do inside a class from what you do outside a class.
- Separate exposed behavior from internal behavior.
  - Mark all instance fields private
  - Initialize instance fields in the public constructor
  - getter & setter methods

# Midterm 1 Study Guide 😊

## Unit 1 : Java

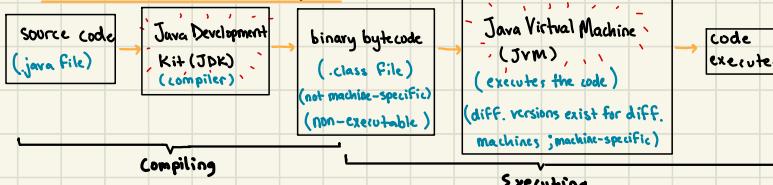
### Interpreted languages

VS

### Compiled languages

- contain built-in language interpreter that parses & interprets src code for execution
- same src code can run on diff platforms & processors ✓✓
- runs slower than compiled XX
- Ex: Python, Javascript
- There is a machine-specific compiler that parses + translates src code into machine-executable code
- lower-level ; speaks directly to machine
- runs faster, can be highly optimized //
- Machine-specific versions of the code needed for every kind of processor (not universal) XX
- Ex: C, C++, Rust

→ Java : Best of Both Worlds !!



### Executing large projects

→ Dependency: 3<sup>rd</sup> party library (like JavaFX or JUnit) of code used in our project

→ At runtime, all dependency & source code files get compiled/packaged into 1 .jar file (a bunch of .class [bytecode] files archived into one.)

## Unit 2: Object-Oriented Programming

- flips the relationship between input & the function performed on it.
- collecting data together as an abstraction so that we can work w/ that data (is like the main class) w/o having to know the specifics of how that data is maintained, interpreted, & used. — much easier
- **Abstraction:** provides a means to invoke "behavior" & save objects of the class type.

### Steps to OOP

- ① Name the abstraction : `Circle.java`  
`public class Circle {`
- ② declare its fields:  
• "private" b/c of encapsulation  
• NOT inside any method
  - `private double radius;`
  - `private int centerX;`
  - `private int centerY;`
- ③ define a constructor : `public Circle (int centerX, int centerY, double radius) {`  
 `this.centerX = centerX;`  
 `this.centerY = centerY;`  
 `this.radius = radius;` }

- ④ define instance methods;  
• execute instructions to return a value or behavior based on info from the fields.  
• called using `referenceObj.method();`

→ **memory:** Whenever we run a line of code invoking a constructor, Java will set aside the amt of memory needed to store it — in the heap

\* After doing this, Java actually starts the constructor.

→ **"static"**: non-object-oriented; global & can be called by anybody

→ **class members**: The methods & fields of a class. But NOT the constructor.

- They define values & helper methods assoc. w/ the class as a whole.
- have to be defined by the **static keyword**... Otherwise it's an instance field/method (member)
- constant values.

→ **Object**: Each object is an instance of the class, & the object type is the name of the class (eg a `Circle` object)

→ **Keyword Final** on a ...

- Method - method cannot be overridden by a subclass
- field - value of field/variable cannot be changed after the constructor instantiates them
- class - class cannot be extended (have subclasses)

### → Keyword this:

- Keyword inside constructor that points to new object to be initialized
- a memory pointer/reference to the memory space that Java set aside for the new object (when the **new** keyword was used to invoke the constructor)
- tells computer where to send the data that was passed in through the class fields.
- If class field names DO NOT OVERLAP with the names of any local parameters or variables in the constructor/methods... then **this** can be omitted.

| Static methods                                                                                                                                                                                                                                                                                                                              | VS | instance methods                                                                                                                                                                                                                                                                                                                                                            |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• general, public methods</li><li>• not instance-specific</li><li>• referenced through the <b>Class</b> name that they are in ;</li><li>• <code>double side_ab = Triangle.pointDist (...);</code></li><li>• <small>(...);<br/>name of class<br/>static method's parameter arguments</small></li></ul> |    | <ul style="list-style-type: none"><li>• only make sense when called through a particular instance</li><li>• referenced through the <b>instance object's name</b> ;</li><li>• <code>Triangle testTri = new Triangle();</code></li><li>• <code>double area = testTri.getArea();</code></li><li>• <small>name of specific object<br/>name of instance method</small></li></ul> |

→ See "Anatomy of a Class" section of the review session notes.

## Unit 2: Encapsulation

- first see "Encapsulation Principles" section of the review session notes.
- **getters**: public method used to retrieve value/data from a class' (private) field.
  - Users can access data w/o having direct access to the class fields.^
  - ALWAYS use them, even if you are making your fields public for some reason.
- **getter trick-derived fields**: an imaginary "field" (in that it is a data value that's relevant to your abstraction) that is actually just a calculation or transformation of other fields.
  - don't declare it as a class field or ask for it in the constructor args; instead, just write a getter method for the "field" & perform the calculation directly inside the method.
- **setters**: public method that sets+updates a field;
  - `public void setLength(int length) { this.length = length; }`
- **setter validation**: throwing an exception (in an if-statement) if the user tries to set a field to an improper value (for ex, a negative int for a "length" field.)

## Unit 2: Interfaces

- ADT that provides a list of methods that any implementing class is promising to provide a coded implementation for - sort of a contract.
- Publicly declared, AND implementing classes must declare all impl methods as public.
- Several classes can implement 1 interface AND a class can implement multiple interfaces.
- Only 2 other things allowed inside Interfaces:
  - ① static methods (since they don't need a specific instance's data to be implemented).
  - ② default methods - instance methods that can be implemented entirely using other methods of the interface (and no fields). EX: `return get(x) - other.get(x);`
- \* Implementing classes have choice to use the default method or define their own impl!
- programming to the Interface: store objects as the Interface type e.g.  
`Yarn leverunner = new YarnImpl`

→ Encapsulation: separating an abstraction into 2 parts:

- 1) Interfaces
- 2) Classes

## Unit 3: Inheritance

- Superclass: regular class (in terms of code)
- Subclasses: extend the superclass (inherit all of its methods & fields)
  - They can also add their own extra fields/methods.
  - Subclasses automatically 'contain' all superclass members unless we use "extends"
  - Subclass constructor: use super (...) as 1<sup>st</sup> line of constructor to essentially "call" the parent class constructor (as if it were a method)
    - inside parentheses: same parameter args that are required by pc constructor.

```
public Person (String name){ }
this.name = name; }
} : public Student (String name){ }
super (name); }
```

→ Subclasses in memory: both references point to the same memory address in heap:

```
Person avi = new Student("avi");
(this is ex of subtype polymorphism)
```

→ What difference does it make in the reference used to declare a subclass object?

- User only has access to the methods defined for an object's declared type (aka the reference) - `Person avi = new Student("avi")` is declared as a `Person` type & thus can't access any methods that are specific to `Student.java`

\* Object avi = new Student("avi") can ONLY access the Object class'  
"equals" and "toString" methods.

→ Multiple inheritance: when a class extends more than 1 parent class  
NOT allowed for classes, but IS allowed for Interfaces

## → Inheritance with Interfaces

- provides method signatures for its parent Interface AND "grandparent" Interfaces
- often used for picking a few methods across several Interfaces and pulling them all together

# Unit 6: Error Handling

What is an exception?

- Software controls hardware, & mistakes in computer software can be dangerous.
- an unexpected, unusual, or abnormal situation which arises during execution of a program.
- some can be anticipated by the programmer & thus dealt with by the program (e.g. writing code to "throw" exceptions) ... other times, they can cause the program to crash.

- History: early error handling strategies -

① What is a "global error code"?

- A global variable (e.g. `public` and `static`) that has its own special spot in memory, and where an "error code" is stored to indicate if something has gone wrong.
- for ex, declaring this at the top of a class:  

```
public static int error_code;
```
- Whenever something goes wrong, we have code that changes `error_code` to some other numerical code.
  - \* Prior to this, programmer has to know/define a list of errors & the codes that correspond to them.
- Anytime the code/program does something where an error could occur, we have to check the error variable to see if it is still = 0 (indicating no error), or if it has changed to a diff value.
- It is on the programmer to know all of these codes & what they mean.
- It is on the programmer to remember to write code 'checking' for an error at any place where one could occur.
  - \* Otherwise, the program could just continue on unaffected, causing bigger issues to arise later.

How do we use that variable?

② What are the issues with global error codes as an error handling strategy?

- Have to 'clear out' the variable's value everytime after an error has been handled.
- Only 1 variable; if a second error occurs while the 1<sup>st</sup> one is being handled, there is nowhere to store its code.
- if a function returns some out-of-range value that it should not have produced, then there is a designated special return value that is meant to be interpreted as an error.
- All methods that we would normally declare as `void` (because they are procedural & don't need to return anything) would instead not be void functions & would return a number indicating the error status.
  - \* for ex, 0 = success, < 0 indicates error ... & different neg. values correspond to diff errors (as documented).

How do these work with void methods?

How do these work with normal methods?

③ What are the drawbacks to these early error handling strategies?

- Inconsistent & convention-based
- Methods must have out-of-range values to use for indicating that an error occurred.
- Relies on documentation (created by the program's creator) to explain what each error means -
  - \* And this documentation needs to be well understood by any other programmer using this program.
- The programmer is responsible for remembering to check for errors.
- Difficult to extend in future development
  - \* e.g., including different errors not initially coded into the solution, new features providing info about errors, etc.

# Exceptions - the 'modern error handling strategy'

What is exception handling?

- A formal method for detecting, signaling, & responding to errors.
- Every programming language (except C b/c it's the oldest) provides a built-in mechanism for exception handling.

What are the benefits of exceptions?

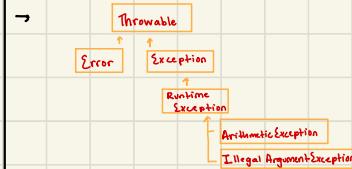
1. Consistent, extensible, modular
2. Expressive; can express exactly what type of error occurred, and can **encapsulate details about the error**:
  - the line of code where the error occurred, what type of error, etc.
3. Dependable, obvious behavior:
  - if an error occurs, the programmer knows about it & can decide whether to handle it.
4. Safe:
  - programmer can designate certain pieces of the code as being **critical**, and have this code run & execute no matter what, even if an error occurs
  - built-in exception handling allows us to do this.

So what exactly do "exceptions" look like in Java?

- They are **represented by objects!**
- Every specific type of error gets its own "exception" object type (aka 'class')
  - these objects use **encapsulation** to store details about the instance-specific error that just occurred.
  - they also **use inheritance to classify the kind of error that occurred.**
- Yes, for our specific needs we can create new exception classes.
- **BUT! Java also provides built-in exception classes for common errors**
  - for ex, `IllegalArgumentException`, `FileNotFoundException`, `IOException`

What else does Java provide?

- The **inheritance (parent & sub-classing) framework** for all exception classes.
- There are various subclasses & sub subclasses used to classify errors in Java
- The **Throwable** class — the parent class to all exception/error object classes.
- Any new exception object we make must also inherit/extend **Throwable**.



What are the 2 parts of exception handling?

- **Throwing** and **catching**
- the "**detection**" aspect of error handling — signaling that something has gone wrong.
- the "**handling**" aspect — dealing with the error itself.

## - Throwing an exception -

What is the syntax?

- use the **throw** keyword;
- `throw new [name of exception class] (encapsulated values specific to exception);`
- `throw new IllegalArgumentException ("null value provided.");`
- Usually, the exception object is created at the time that it's being thrown. (hence the **new** keyword).

What happens when an exception is thrown?

- As soon as the line of code with the `throw` occurs, the method stops executing.
- We start "unwinding the stack" and looking for the handling mechanism to deal with the (now detected) error - aka looking to see if the current method is inside a try-block.
- One of 2 things will happen next:
  - a) the program 'unwinds' & finds the 'catch' method which then handles the error; program continues to run & user isn't even necessarily aware that an error occurred.
  - b) the program 'unwinds' (past the current method and) all the way back to `Main.java` (where the current method was initially called)...if error still isn't handled, the program stops/crashes & gives the user an error message.
- exceptions force the issue by throwing an exception & threatening to stop the program if the error isn't handled.

### - Catching an exception -

What does "catching" entail?

→ Writing/providing the code to handle an error/exception.

What are the 2 parts of the 'catching' code?

① try blocks

② catch blocks

What is a try block?

→ A block of code (indicated by the `'try'` keyword and a block in curly brackets) where we write the code that has the possibility of throwing an exception.

• for example, calling the methods of the program which contain `throw` statements.  
`try {  
 methodB();  
}  
3`

- When an exception is thrown in a method, the first thing the program does while unwinding is look to see if the method (as a whole) is inside of a try-block.
- Once the computer finds the method inside a try-block, "unwinding" stops & we immediately go down to the first catch block.

What is a catch block?

→ a block of code (indicated by the `catch` keyword and a block in curly brackets) that contains the actual code for handling a given exception.

→ There are (usually) several catch blocks - a single catch block corresponds to a single type (class) of exception

→ FORMAT: `catch (ExceptionType e) {`

// code to respond to this specific ExceptionType 3

`catch (OtherExceptionType f) {`

// code to respond to this other ExceptionType 3

What is the call stack?

→ where the program keeps track of all active method calls & the order

→ [EX] imagine the methods `getWeight()`, `getTitle()`, and `printDescription()` in a yarn inventory program.

• `public String getTitle() {`

return this.name + this.getWeight().toString(); 3

```
• public void printDescription() {
 System.out.println ("The title of this yarn is " + yarnObj.getTitle()); } 3
```

→ Now in the `Main.java` file, imagine we have created a new Yarn and run the following line of code:

```
myNewYarn.printDescription();
```

- Main is calling `printDescription()` which in its code, calls `getTitle()`
- `getTitle()` then calls `getWeight()`
- `getWeight()` calls `toString()`

→ As these methods are being called, the call stack gets populated & then depopulated ... until `toString()` is fully executed, the rest of the methods are still "open" because they haven't been fully executed ... during this time, the call stack might look like this (abstractedly):

```
printDescription
getTitle
getWeight
toString
```

→ ...so for the purpose of understanding exception handling, we can basically think of call stack as a list of active method calls (stored somewhere in memory).

→ Take this example... imagine that `methodA()`, `methodB()`, and `methodC()` all contain `throw` statements.

1. `methodA()` is called by main, everything goes smoothly,

no exceptions thrown.

2. `methodB()` is called by main & an exception gets thrown.

Once the exception is thrown, `methodB()` immediately stops executing.

3. `methodC()` will never be executed because the exception occurred up

in `methodB()` - even if/after the error is handled!

4. Now we are unwinding the call stack & searching for where

`methodB()` was called & if it was called inside a try-block.

5. Since we found the current method (`methodB()`) to be inside a try-block, we now immediately jump down to the first catch block.

5a. If `methodB()` hadn't been inside a try-block: call stack unwinds to the previous method that called `methodB()`

in the first place... it stops the execution of this method & again searches for a try block

\* This process continues to happen - moving up levels - until a try block is found.

6. Check to see if There is an is-a relationship between the exception that was thrown & the exception being declared in this catch box. If there is, then the code inside this catch block is executed.

→ Catch blocks are similar to if-else statements - once we find a match, we ONLY execute that block. The succeeding blocks don't even get looked at.

6a. If there isn't an is-a relationship, we move on to the next catch block, & so on.

```
try {
 methodA();
 methodB();
 3. methodC();
 4. 3 catch (ExceptionType1 e) {
 (...) 3
 catch (ExceptionType2 f) {
 (...) 3
```

What do you mean by **is-a** relationship  
in regards to exceptions?

Example ?

→ Remember that exceptions are objects! Recall the Java exception class hierarchy...

- The exception being thrown might be a hyper-specific subclass of a broader category (parent class) of exceptions... The catch block could refer to a parent exception type & not necessarily the specific subclass type.

methodD() {

    throw new **IllegalArgumentException("lol.");**

}

method  
in  
program

try { methodD(); }

catch (**RuntimeException e**) {

    (...)

error  
handling  
code

→ **IllegalArgumentException** is-a (is a subclass of) **RuntimeException**

→ Ask this: is [exception being thrown] is-a [exception type declared in catch block] ?

→ After looking at all of the catch blocks corresponding to the current try-block without finding a match in object type, then we continue back to "unwinding the stack" and looking for try blocks in the method calls that preceded the current one.

→ A.k.a., we are "re-throwing the exception".

→ If we never find a match: program dies, error message displayed on screen.

→ Wait... everything in the try block gets executed??!

→ If a catch-block for a certain exception "A" is placed after a catch-block for an exception type that is a parent class "B" of "A", then this is bad because the "A" catch block will never be executed!

↳ aka  
"A extends B"

immediately

- all thrown exceptions of class "A" will enter into the catch block for "B", which wasn't our intention
- the compiler can notice & inform us of these errors.

→ the same program where the methods are being called! For instance, in **Main.java**.

- they don't exist in a separate file... the concept is basically that everytime you are writing code that calls a method that contains a **throw statement**, you want to call that method inside of a try-block rather than just on its own (like we've been doing so far) ... so that we can specify a response to the "Throw."

→ COMPARE

public static void main(String[] args) {

    int num = yarn1.getWeight();

}

VS

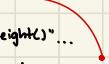
public static void main(String[] args) {

    try { int num = yarn1.getWeight();

}

    catch (ExceptionType e) {

        //some specified action



→ both of these files are attempting to do the same thing - declare int "num" as "yarn1.getWeight()"...  
but if the **getWeight()** method contains a **throw statement**, we should be following the 2nd format.

\* note: we are only talking about unchecked exceptions thus far

## - the "finally" block -

What is a "finally" block?

→ block of code indicated by the **finally** keyword & curly brackets

→ place this at the end of the sequence of try- and catch- blocks.

**finally { ... }**

→ a place for code that needs to be executed no matter what — whether or not exceptions are thrown, whether or not they are handled.

→ goes back to the idea of **critical** code — code that should run no matter what.

→ simply create a class that extends from one of the built-in Java exception classes.

→ Example:

```

the public class NotCoolEnoughException extends RuntimeException {
exception ...
class ...
}
throwing if (num <= 10) {
the throw new NotCoolEnoughException(); }
exception

```

→ **throw** : used to throw an exception object

→ **try/catch** : used to safely run code that might throw an exception.

→ **catch** : blocks similar to if-else statements ; need to be ordered from most to least specific object type

→ **finally** : used for code that must always execute

• usually used for cleaning up & closing system resources.

1. Throw exceptions early ... as soon as you detect a wrong value

• this is defensive programming

2. Be specific when throwing an exception

• try to use a built-in type but don't be afraid to make your own to describe a situation.

3. Catch exceptions late.

• Just b/c you can catch an exception doesn't mean you should

• Don't catch an exception unless you know how to deal with the situation

• Instead, let the exception "bubble up" to a level of the program where it will actually make sense.

→ You don't want to catch an exception just for the purpose of catching it — want to catch it because we actually have some programmatic way to deal with it

• This usually isn't going to be at the same point where the exception occurred, and instead will be at a much higher level in your program.

• therefore, you should let the exception unwind the stack until a point where its reasonable to catch the exception.

→ If an exception is thrown at a point where you wouldn't know what it means in respect to the larger program, then that's probably NOT the right place to catch it.

## Summary of terms?

What are some "best practices"

with exceptions ?

→ **throw** : used to throw an exception object

→ **try/catch** : used to safely run code that might throw an exception.

→ **catch** : blocks similar to if-else statements ; need to be ordered from most to least specific object type

→ **finally** : used for code that must always execute

• usually used for cleaning up & closing system resources.

1. Throw exceptions early ... as soon as you detect a wrong value

• this is defensive programming

2. Be specific when throwing an exception

• try to use a built-in type but don't be afraid to make your own to describe a situation.

3. Catch exceptions late.

• Just b/c you can catch an exception doesn't mean you should

• Don't catch an exception unless you know how to deal with the situation

• Instead, let the exception "bubble up" to a level of the program where it will actually make sense.

→ You don't want to catch an exception just for the purpose of catching it — want to catch it because we actually have some programmatic way to deal with it

• This usually isn't going to be at the same point where the exception occurred, and instead will be at a much higher level in your program.

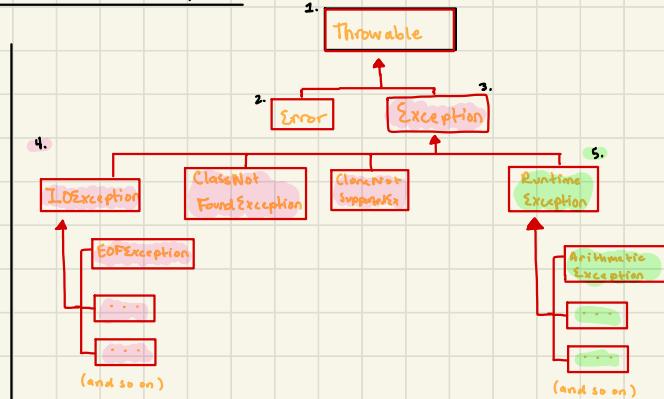
• therefore, you should let the exception unwind the stack until a point where its reasonable to catch the exception.

→ If an exception is thrown at a point where you wouldn't know what it means in respect to the larger program, then that's probably NOT the right place to catch it.

Why should we "catch" exceptions as late as possible?

## - checked versus unchecked exceptions -

What is Java's built-in class hierarchy for exceptions?  
(in more detail)



1. "Throwable" is the superclass of all exception objects
2. "Error" represents externally caused, unrecoverable problems that should generally not be caught or handled.
3. "Exception" is the superclass for errors caused by the program itself, which may be caught & handled if appropriate.
4. Checked exceptions - "Exception" & all of its descendants EXCEPT `RuntimeException`
5. Unchecked exceptions - "`RuntimeException`" and all of its descendants

- Exceptions for situations that are usually caused by factors inside the program's control
  - For ex, they might indicate logic errors or an unexpected variable value.
- These are for errors that really "never should have happened", e.g. the fault of the program / programmer
- We should only use unchecked exceptions to handle these types of errors **if we know how to fix the situation.**
- Exceptions for errors that are usually caused by factors outside of the program's control:
  - They may occur even during "normal" operating conditions.
  - Errors that may indicate an unexpected system state.
- They are for errors that your program should probably anticipate & be able to respond to
- checked exceptions are subject to the **"catch or specify"** rule! (unchecked exceptions aren't)
- The distinction between unchecked & checked is a Java-specific feature, & was created to account for "programmers being lazy" & not checking for external errors when they should be.

What are checked exceptions?

What is the main difference between checked & unchecked exceptions?

Where do we throw and catch checked exceptions?

- Inside of the method itself!
- As opposed to unchecked exceptions, which are thrown inside the method BUT caught & handled in the "main" file where their respective methods are actually being called
  - aka what we just learned (see previous pages)

What is the "catch or specify" rule?

- The format that all checked exceptions must follow
- If a method contains code that might throw a checked exception, then the method must do one of the two:
  - ② Catch the exception internally
  - or ② Specify that the checked exception might be thrown by the method — this is done in the method signature

Memory tool for remembering "checked" versus "unchecked"?

- "unchecked exception" = didn't 'check for' account for errors yet = try & catch in the main file, as we are calling the methods.
- "checked exception" = already 'checked for' errors by the time we get to main file = catch & specify in the method itself so we can consider it 'already checked'

Why do we follow the "catch or specify" rule?

- It is defensive programming — Forces the programmer to address the situation.
- Even though checked exception errors are out of our control, it's better if our program defines a strategy for dealing with those situations.
- If the current method is the correct place to deal with the error, and we know how to deal with the error at this level... catch the exception.
- If this current method isn't the appropriate place to deal with the error, needs to be dealt with at a higher level, (recall the principles of throwing) ... specify the exception.
  - \* specifying an exception is basically a way to warn anyone who calls the given method that they are going to have to deal with the error themselves, if it occurs.
- The class with the method that has potential for errors:

```
public Scanner openFile(String filename) {
 File f = new File(filename);
 Scanner s = null;

 try {
 s = new Scanner(f);
 } catch (FileNotFoundException e) {
 e.printStackTrace();
 } return s;
}
```

even though there isn't an explicit throw statement anywhere (like with unchecked exceptions), there are exception throws built into Java's "Scanner" class (you can see this in the 'Scanner' documentation).

try- and catch-blocks, just like with unchecked exceptions.

example of specifying a checked exception?

```
public Scanner openFile(String filename) throws FileNotFoundException {
 File f = new File(filename);
 Scanner s = new Scanner(f);
 return s;
}
```

the method signature (aka the line where a new method is created) is specifying that this method might throw a "FileNotFoundException" exception

- If we simply specify an exception in a method & don't do anything else, an error will occur when the method is called!! Because we haven't handled it anywhere!!

What happens (to the prog as a whole) when an exception is specified?

So then what else do we have to do?

(continuation of ex  
on previous page)

→ This is known as a "catch or specify error"

→ In the main class where the method that calls the method w/ the 'specified exception', either:

② handle the exception by calling the method in a try block:

Main.java

```
public static void main (String [] args) {
```

```
 try { openFile (aviFile); } ← calling the method that specified an exception
```

```
 catch (FileNotFoundException e) {
```

```
 // code to handle error
```

```
}
```

{OR}

③ have the current (main) method ALSO declare/specify that it could throw an exception (essentially instructing the error to continue "bubbling up"):

Main.java

```
public static void main (String [] args) throws FileNotFoundException {
```

throwing exception in main

method signature, so try/catch → openFile (aviFile);

blocks not needed

```
}
```

→ At compile time... if you call a method with a specified exception & don't write any code to accommodate this, your compiler won't let you run the code.

### - Compile-time v.s. Run-time errors -

What are compile-time errors?

→ errors that are caught before you run the code - for ex, IntelliJ will give you red underlines in your code.

→ essentially because the compiler can't read/understand your code

→ Syntax errors; unbalanced brackets, missing return statements, missing class definition, etc.

→ Static Analysis - errors that can be identified simply based on the "static" text... don't need to run the code to find them.

- unreachable code, "catch or specify" violations

→ IntelliJ might warn us, but usually you have to run the program to see. The compiler can't tell that anything is wrong.

→ All exceptions! e.g.; Throwable class objects - Error, Exception, and all of its subclasses.

Compile-time errors indicate...

something is incomplete about your program.

Runtime errors indicate...

something is wrong with the logic of your program.

What types of errors occur at run-time?

Key difference between the two?

## Unit Testing and JUnit

What are the 4 levels of professional software testing?

1) Unit Testing

→ From lowest to highest level:

→ Testing methods and classes in isolation. To ensure that that specific class is working correctly.

→ This step occurs during development.

→ Testing how new code integrates with existing modules.

→ "Does my new code break my existing code?"

→ occurs during development

→ Testing the entire system as a whole.

→ occurs after development

→ Test in a production-like environment.

→ "Does my code work in realistic conditions?" Testing it on different machines/computers etc.

→ As opposed to Systems Testing, which is in the "ideal environment" of just your computer.

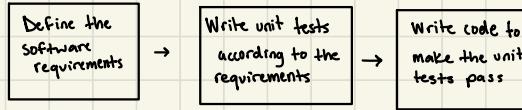
→ occurs before release of a software.

→ A principled approach to use when building smaller modules of code.

→ In the most extreme following of TDD, you should write your tests even BEFORE you write your code.

What is test-driven development?  
(TDD)

What are the outlined steps of TDD?



→ In reality, all of these steps happen kind of simultaneously... you go back and forth

### - JUnit -

What is JUnit?

→ a library/framework (in Java) to help us write unit tests.

→ extremely well-known in the Java world.

→ A library of **assertion methods**:

• these allow us to make statements (assertions) about what should be true at some point in your test

• the assertions will either confirm that, or will run/raise some exception.

→ The **@Test annotation**:

• a compiler directive for us to mark which methods inside a test class are intended to act as individual JUnit tests.

• the **assertion methods** are what we use inside of these JUnit tests.

→ A user interface & other tools that provide the ability to **automate testing**.

→ After you run the test class: a report of which tests worked & which failed.

→ we add JUnit as a dependency in Maven, and then we can add imports for JUnit features such as:

`import org.junit.Test;`

`import static org.junit.Assert.assertTrue;`

How do we access these JUnit features?

**tc = test class**

**pc = "program class" — the class which is being tested**

What are test classes?

→ a separate set of classes where we test the classes we are writing for our program.

What goes in a test class?

\* \*

- Conventionally, should have a **separate test class for each class of the program**.
- One or more **test methods** which actually test the class — all of these are marked/annotated by the **@Test** compiler directive.
  - every test method checks a single method, field, or constructor of the class being tested.
- Optionally can also contain other stuff, such as helper methods that might be used by your test methods... but make sure to distinguish these by **only** adding **@Test** to test methods.
- **No.** For JUnit to create its report, it **creates an instance of the test class & then calls each of the test methods & detects whether they [A] work or [B] throw an exception.**
- the idea behind a test method is that it writes code to exercise some feature, and then either throws an exception, or it runs smoothly and just works
- the return type is usually **void**, b/c JUnit just wants to know whether or not it will run顺畅ly.
- usually we create an instance of the **pc** object & test a particular part of the pc, comparing what we expect to happen, to what actually happens... this is where JUnit's **static assertion methods** come in.

What does the body of a test method generally look like?

- The static assertion methods (within the tc methods) run smoothly if the expected output matches the tc method's output... and if not, they raise an exception.
- If the code throws an uncaught exception, the test fails
- If it doesn't, the test passes
- The JUnit dependency incorporates several JUnit tools & interfaces into the IDE (IntelliJ), such as
  - a button to run all the test methods in the tc at once
  - a button to run just one test
  - a report of the results (pass or fail) returned to us in the console after every run.

### - Writing a Unit Test -

So what code goes inside a unit test?

→ it depends on what you're trying to test, but a typical JUnit unit-test does this:

1. create an instance of the pc
2. use some methods to change that instance's internal states or otherwise do something with it
3. Use JUnit assertions to verify that the instance methods return the correct values.

What is **assertTrue()**?

- One of JUnit's assertion methods that takes in **any condition** as its parameter & checks to see whether that condition equates to true... a very simple concept, but representing it in the form of one of JUnit's **assertion methods** is how JUnit is able to derive results from tests.
- if the condition inside the () is true, the test passes. If it is false, **assertTrue()** throws an exception and the test fails.

**assertTrue ( condition statement )**

What is an example of a unit test?

→ We have an `InventoryImpl` class that is supposed to start each instance off with a `capacity` (field)

of 0 ... so we are testing the constructor of `InventoryImpl` to make sure it does that:

```
① @Test
② public void testCapacity() {
③ Inventory inv = new InventoryImpl();
④ int capacityTest = "inv.getCapacity();
⑤ assertEquals (capacityTest == 0);
⑥ }
```

what "assert True" says

⑦ "I want to assert this statement as being true"; that the value of `getCapacity` retrieved from the new `Inventory` object is equal to 0.

→ if the expression passed into `assertEquals` is true, the JUnit test passes. If not, it fails.

What are some of the other assertion methods JUnit provides?

| Method                                            | Description                                                                                                                                        |
|---------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>assertTrue(condition)</code>                | throws an exception if condition is false                                                                                                          |
| <code>assertFalse(condition)</code>               | throws an exception if condition is true                                                                                                           |
| <code>assertEquals(expected, actual)</code>       | throws an exception if "actual" is not equal to "expected"<br>(uses <code>.equals()</code> to check equality, aka CONTENT equality)                |
| <code>assertNull(object)</code>                   | throws an exception if object is not null                                                                                                          |
| <code>assertNotNull(object)</code>                | throws an exception if object is null                                                                                                              |
| <code>assertSame(expected, actual)</code>         | throws an exception if objects are not the same reference<br>(if "expected" and "actual" aren't the same object in memory, aka REFERENCE equality) |
| <code>assertNotSame(unexpected, actual)</code>    | throws an exception if "unexpected" == "actual"<br>(if they are the same obj. in memory)                                                           |
| <code>assertArrayEquals(expected, actuals)</code> | throws an exception if the arrays do not contain the same elements (using <code>.equals()</code> ) aka (CONTENT equality)                          |
| <code>fail()</code>                               | throws an exception to fail the test                                                                                                               |

When is the `fail()` method useful?

- For more complicated tests that retrieve several different values ... as soon as you get a value that isn't what it should be, you can assert `fail()` and end execution of the test method.
- Similar concept to `if`-statements that have `return` statements built into them ... if the computer falls to a line inside an `if`-block that says `return [x];`, it returns that value & ends execution of the method, even if there are more lines of code below.
- Yes, if you wanted to, you could always just boil your tests down to that
- however, having higher level semantics associated with your assertions can be useful.
- because the more semantically meaningful an assertion is, the more detailed the information that JUnit reports to us will be - this is useful especially when a test fails.
- use the most specific assertion possible because it describes the situation more fully.

But couldn't we always just use `assertTrue()` or `assertFalse()`?

Why is it better to use more specific assertions when possible?

EXAMPLE: What would be a better assertion method for the `testCapacity()` method example (prev. page) ?

→ Sometimes IntelliJ will even notice these potential improvements (in assertion method choice) and suggest them to you.

@Test

```
public void testCapacity() {
 Inventory inv = new InventoryImpl();
 int capacityTest = inv.getCapacity();
 assertEquals(0, capacityTest);
}
```

→ in this case, `assertEquals()` is semantically more meaningful than `assertTrue()`... if the test were to fail, JUnit would give us a much more useful report of what occurred:

assertTrue(capacityTest == 0);

"something that was supposed to be true was false."

VS

assertEquals(0, capacityTest);

"you were expecting the value 0 but got [...] instead."

more helpful/detailed when it comes to fixing our code

## Unit Testing in Formal Software Development

What are the steps in the beginning stage of software development?

1. Start with an algorithm specification

2. Write unit tests:

- code that runs through anticipated "normal" situations as well as "abnormal" (edge) cases

3. Write an implementation;

- Implement the algorithm according to the specification

These 2 steps may happen in any order

4. Once the implementation passes the unit tests, the code is ready for the next stage of the software life cycle.

→ The step where you:

- Design an interface with carefully chosen methods
- Discuss use cases with stakeholders
- Determine desired behavior for edge cases
- Make sure you clearly understand how the algorithm should work
- Walk through a few executions so everyone agrees on what should happen

→ This is the step where we write the documentation for the algorithm (via an interface, for example)... How do we expect the object and/or algorithm to work? What do we want it to do?

\*\*\* !!

→ pro tip for coding interviews: never skip the algorithm specification step! Take ample time to understand the algorithm

# Unit Testing in Isolation

What is the goal of a single unit test?

- although each unit test should aim to test one aspect of a class, it's hard to test a single method completely in isolation.
- try to isolate unit tests as much as possible, but it's okay to call multiple methods in a test
- **Solution:** write multiple unit tests.
- It is impossible to write tests to verify our program works for every situation, but there are options to help us be as correct as possible - 2 generalized "solutions";
- Considering the algorithm from a mathematical perspective, tracing all execution paths, & proving that the result is correct for every possible machine state.
- This is 1 solution for 'ensuring correctness' that is more time-consuming but necessary for critical applications
- This approach is not going to be covered in COMP 301

(1) What is "formal verification"?

(2) What is the 2<sup>nd</sup> solution/approach?

What is **test coverage**?

How do we measure the **test coverage** of a test class?

## High Test Coverage

- many tests were written to test a variety of expected & edge cases
- tested as many possible diff situations as we could think of
- maximum (ideally all) lines of code in the pc are executed

## Low Test Coverage

- few tests written, all edge cases not covered
- few lines of pc code that get executed, aka lots of holes

## Example: Envisioning a Software and Writing Unit Tests

What is the prompt/task being given?

"We need an object that represents an integer which can be increased or decreased by any amount"

What is the first step?

"We need to be able to test if the current number is prime or not."

→ **Algorithm specification!** First step is to design an interface

- we need a way to add an int value to our object – a `void addValue(int value)` method
- we need a way to test whether the object value is a prime number – a `boolean isPrime()` method

→ Our defined, thus far unimplemented interface :

```
public interface PrimeCounter {
 void addValue(int value);
 boolean isPrime();
}
```

\* we have defined what it means to be a `PrimeCounter` object

What is the next, second step ?

→ 2 options: Either we could start writing the `PrimeCounterImpl` class... or we can first work on creating unit tests for an imagined `PrimeCounterImpl` class

→ the **Test Driven Development** approach says to begin by writing unit tests!

How do we write our first unit test?

→ Start at the baseline... lets write a test to check ...

- that we can create a new `PrimeCounterImpl` object – aka that the constructor works without error
- that `isPrime()` works without error
- that `isPrime()` is initially false.

→ RECALL: follow the steps of a "typical JUnit test":  
 1) create an instance of the class;  
 2) call methods to change the state;  
 3) Use JUnit assertion methods to verify.

```
public void PCI Test01() {
 PrimeCounter pc = new PrimeCounterImpl();
 assertFalse(pc.isPrime());
}
```

throws exception if  
 "pc.isPrime()" = true

How is writing unit tests a helpful step in writing a class' implementation?

→ it reveals to us missing/not fully thought out parts of our design that we should revisit. For ex, this

1<sup>st</sup> test already raises some questions about the design of our object:

{ What is the initial value of our object? Does our constructor need to ask for one from the user (as a parameter)? }

→ While writing unit tests, we will often flip back & forth to the 'design' phase, as we are forced to think more about the design of our object.

→ **add to documentation:** `PrimeCounterImpl` object starts off with a value of 0.

→ if a method gave the correct value in one test, how do we know that its actually working correctly, or just returning that same value for every instance of the class?

- Next step: write a test that should return the opposite value of our first test of a given method.

What should we consider for our next unit test?

→ Question: is `isPrime()` returning 'false' for any value? What if we test a value that is a prime number?

What does our 2<sup>nd</sup> test look like?

```
public void PCI Test02 () {
 PrimeCounter pc = new PrimeCounterImpl ();
 pc.addValue (11);
 assertTrue (pc.isPrime()); 3
```

throws exception if  
"pc.isPrime() == false"

What questions do we still have to consider (and write unit tests for)?

→ Does `PrimeCounterImpl` handle negative values?

```
public void PCI Test03 () {
 PrimeCounter pc = new PrimeCounterImpl ();
 pc.addValue (-3);
 assertFalse (pc.isPrime()); 3
```

→ What if `isPrime()` is called multiple times?

→ What if `addValue()` is called multiple times?

```
public void PCI Test04 () {
```

```
 PrimeCounter pc = new PrimeCounterImpl ();
 pc.addValue (11);
 assertTrue (pc.isPrime());
 pc.addValue (3);
assertFalse (pc.isPrime()); 3
```

These statements are written based on  
what we expect the value to be...eg  
when PCImpl's value is 11, it is a  
prime number. When value is 11+3=14,  
it is not a prime number

# Design Patterns

What is a design pattern?

- A classic approach for solving a common problem that arises when writing code.
- by learning about design patterns, you can recognize when a situation fits one, & then know what the appropriate solution is.
- or you can use one as a template/starting point for your solution.
- the book "Gang of Four Design Patterns" describes common object-oriented design patterns & breaks them into 3 categories: creational, structural, & behavioral
- patterns related to creating new objects
- patterns related to objects interacting with each other.
- common algorithms that are encountered in an OO setting

What are creational patterns?

What are structural patterns?

What are behavioral patterns?

What are the 7 design patterns we will cover in this class?

## creational

- Abstract Factory
- Factory Method
- Singleton

## structural

- Decorator

## behavioral

- Iterator
- Observer

- As well as Model View Controller, which isn't in any of these categories but is used for thinking about a software /program as a whole.

# Iterator

What is **iterator**?

→ A design pattern that "provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation."  
aka a collection

→ We want to sequentially access & loop through every element in a collection.

→ the pattern shouldn't have to deal with or know anything about what the collection actually is or how it is being stored

• Just that it "is a collection."

→ A group of zero or more similar objects; a set of items

→ collections are organized using **data structures** (20!!) ... for example:

• Binary Tree

• array

• ArrayList

• Linked List

• HashMap

→ To loop through the items in the collection, one at a time.

→ For ex., with an array:

```
for (int i = 0; i < arr.length; i++) {
 Item item = arr[i]
 // ... (do something with each item)
}
```

→ this for-loop is **very array-specific**. We need to know a lot about how this specific array **object** works:

• that the index starts at 0; the .length property; the usage of square brackets, etc.

→ we have to go through all of the collection's elements **right here**; we can't loop through a few elements & then go do something else & then come back & pick up where we left off.

→ The operations/methods etc. to loop through each kind of data structure is different - the for-loop example above wouldn't work properly for a HashMap, for example.

→ The need for **data-structure-specific code** for going through a collection

→ huge collections with millions + of elements, where the data is too big to store in memory (can't use an array)

• Ex: users on Instagram

→ generative collections where we sort through a collection that creates items on demand ... basically where we are generating the collection while looping through it.

• there is no finite or initially set size of the collection

• want to be able to loop through such a collection without needing to know the specifics of how the generator works.

→ **Bottom Line:** iterator serves to be a way to loop through collections in general - adaptable to almost any situation.

What is a "**collection**" in computer science?

What is the common operation for looping through a collection?

How is this operation different from iterator?

So what is the need that iterator intends to address?

What other situations does iterator pose a solution to?

What is the idea behind the Iterator object?

So does the iterator object know the details/specs of the collection?

Why is this useful?

What do most programming langs provide (in support of iterator)?

What iterator classes does Java provide?

What does `Iterator<T>` represent?

What does `Iterable<T>` represent?

- to be a level of interaction between the code that is using a collection, & the collection itself.
- For a given collection, a class that encapsulates the details of how to loop through it.
- Yes! that's the point. it contains the details of the data structure, where the data is coming from, how it gets generated, etc.
- because then, we can use this iterator object without having to understand any of the details.
- we create a new `Iterator` class for every particular kind of collection, and it becomes an object that we can use to go through the collection itself.
- We can ask it for some items & then go off and perform other actions & then come back and keep asking it for new items — the `Iterator` object keeps track of where we are in the collection.
- built-in language support for the iterator pattern & a library of interfaces & implementations for most of the common iterator situations
- This means that unless we have a pretty highly specialized collection or very particular way that we want to go through it, we probably won't have to code our own iterator & can just use one of Java's iterator classes.
- 2 Interfaces: `Iterator<T>` and `Iterable<T>`
- Iterator support for all built-in collection data structures, such as `List`, `Set`, `Map`, etc.
- the objects of the collection ; encapsulating the details of how you're going to iterate through each object (this is what we actually use to go through the elements)
  - See the description of "the `Iterator` object" above.
- The classes implementing `Iterator<T>` are regular abstraction/object classes (like `Alphabetizer`) w/ their own purposes of existence — they just also implement iterator methods.
- a collection that is capable of creating & returning an `Iterator` object for its elements, on demand.
- all of the Java collection types are iterable

# The Iterator < Interface

What does the interface look like?

→ `Iterator<T>` is a generic type interface (see page 37 of notes), which means it takes a data type, `T`, as a parameter.

```
public interface Iterator < T > {
 boolean hasNext();1
 T next();2
 default void forEachRemaining (Consumer < ? super T > action) {
 //...3
 }
 default void remove () {4
 //...3
 }
}
```

What is the `hasNext()` method?

→ Answers the question: "are there still remaining items to check?"

2 → Returns the next item in the collection.

→ throws a `NoSuchElementException` after all elements in the collection are seen.

RECALL: what does it mean when an interface has a method marked as `default`?

→ methods for which the Interface actually does provide a coded implementation. Every implementation class of the interface automatically contains that method, BUT the classes have the option to overwrite it & write their own implementation if they want to.

3 → A default method that isn't generally considered essential to the design pattern of an iterator.  
→ provides code that gets applied to every single element remaining in the collection.  
→ not really focusing on this method in this course.

→ A default method that isn't generally considered essential to the design pattern of an iterator.  
→ removes from the collection the last/most recent element that has been given by `next()`.  
→ not really focusing on this method in this course.

## - Using an iterator object -

1. Start with a collection of items:

(Ex) `String[] data = new String[] {"Kappa", "beta", "alpha"};`

2. Create the iterator:

→ 2 ways to do it:

`Iterator < String > iterator = new Alphabetizer (data);`

↓ the class which implements `Iterator < String >`

• the iterator object encapsulates the collection

• even though the `Alphabetizer` class also has its own interface, we want to create this instance as an `Iterator` type because in this case, that's the interface that we are using it for.

OR

3. Use iterator's `next()` and `hasNext()` methods

```
while (iterator.hasNext()) {
```

```
 String str = iterator.next();
```

```
 System.out.println(str);
```

→ OUTPUT:

alpha

beta

kappa

How does the iterator loop differ from an array for-loop (like previous ex.)?

→ All of the array (or other DS) specific details are now hidden. We don't need to know anything about where the data variables are coming from in order to sort them; all we have to do is call `next` and `hasNext`.

→ The code / class that wants to sort their data with `iterator` doesn't need to know anything about how the collection works (i.e. the size of collection, sorting method, etc.) — all of this is encapsulated inside the iterator object.

→ that the collection will not be modified while the iterator is actively being used.

→ Some iterator objects provide special methods like `remove()` for safely modifying the underlying collection.

What does the iterator pattern assume about the collection?

### — Designing an Iterator class —

What does the Iterator object need to be able to do?

- track progress through the collection
- Know which items have been seen, and which are coming up next
- Manage the order of the items Without modifying the underlying collection & its order
- At the bare minimum, an iterator must a) have access to the collection, and b) have a way to track which items have been seen
- There are 3 different strategies for designing an iterator

→ Encapsulate a direct reference to the original raw collection (in the `Iterator` object)  
• **not making a copy of the original data**

→ Add a "cursor" field to track progress through the collection

- for ex, an `index` number associated with a certain element
- or if our sorting method is more specialized (like with `Alphabetizer`), then the cursor is keeping track of several different pieces of info in order to track where we are in the collection.

→ Update the "cursor" field each time `next()` is called.

→ This implementation serves to return the elements of a `String` array in the order that they are indexed in the array — a super elementary sort just for the purposes of understanding

Strategy 1 - encapsulate the raw collection

Example of an Iterator object using Strategy 1 ?

### Strategy 1:

```
public class StringSort implements Iterator <String> {
 private String[] collection;
 private int cursor;
 public StringSort (String[] collection) {
 this.collection = collection;
 this.cursor = 0;
 }

 @Override
 public boolean hasNext() {
 return cursor < collection.length;
 }

 @Override
 public String next() {
 if (hasNext()) {
 String item = collection[cursor];
 cursor++;
 return item;
 } else {
 throw new NoSuchElementException();
 }
 }
}
```

• field for tracking collection progress

• encapsulating the reference to the original collection

using the cursor and the collection length to figure out if there are still items left to visit.

The first thing that next() should do is call hasNext(); if it is false, throw a new exception

retrieve the item that the cursor currently "points to", and then increment the points

What are the advantages & drawbacks of Strategy 1?

Advantages

- Memory efficient since we do not clone the collection.

Drawbacks

- It's hard to change the order of the items.
- this strategy is pretty difficult to implement for iterators that are sorting in more complex ways than simply index order (like in this example).
- There's no defined behavior for if the collection is modified externally.

What is the .sort() method?

- a static method of Java's `Array` class (can be used after implementing `java.util.Arrays`)
- for a given `int[]` or `String[]` array, the line `Arrays.sort([name of array]);`

reorders the elements of the array either numerically or alphabetically.

- Encapsulate a clone of the raw collection (in the constructor).
- Sort or Manipulate the cloned collection to make iteration easier.
  - Now that we have our own copy, we can apply `.sort()` to reorder it without changing the o.g. collection

Strategy 2: encapsulate a clone of the raw collection

Example of iterator using

### Strategy 2 ?

→ follow the same remaining steps as in STRATEGY 1

```
public class StringSort implements Iterator <String> {
 private String[] collection;
 private int cursor;
 public StringSort (String[] collection) {
 this.collection = collection.clone();
 Arrays.sort (this.collection);
 this.cursor = 0;
 }
```

the rest of the class looks the same as the example from  
strategy 1 !

What are the advantages and  
drawbacks of Strategy 2 ?

#### Advantages

- Changing the order of items in the cloned collection doesn't affect the original.
- Changing the order or number of items in the org collection doesn't affect the iterator.
- Convenient approach for iterators that would benefit from sorting the collection.

#### Disadvantages

- extremely memory-inefficient because it requires a full copy of the collection.
- imagine really large collections, with like 1M+ elements
- this also means that if a user wanted to use an iterator just to retrieve the first few elements of a collection, you'd still have to apply `.sort()` to all of them — which uses a lot of memory.
- cannot work for infinite collections.

→ Useful for when you want to build a more complex iteration.

→ Encapsulate another iterator of the raw collection :

- basically build the more complex iterator "on top of" existing simple iterators by calling the simple iterator & then working with its elements
- each time `next()` or `hasNext()` is called, use the simple iterators `next()` & `hasNext()` methods.
- essentially "translating the results" of the other iterator.

#### Advantages

- relies on the other iterator object to do the "hard work"

#### Disadvantages

- Can be tricky to implement
- Only works when there is already an existing simple iterator for the collection.

What are the advantages & drawbacks  
of Strategy 3 ?

# The Iterable<T> Interface

What is the Iterable<T> interface?

- A way for a particular data structure (or any object that represents a collection) to promise that it can create an iterator object.
- All of Java's collection class implement Iterable<T>.
- (Aka an object / class that implements Iterable<T>) ... any class can implement the interface and claim to be "Iterable" as long as it provides an iterator() method which creates & returns an iterator for the collection.
  - this is the only requirement (which you can see from the Iterable<T> class code.)
- Iterable<T> does not make any promises on how the iterator object will work.
- [name of collection].iterator(); {the method that Iterable<T> defines}
- Examples:
  - List<String> myList = new ArrayList<String>();  
Iterator<String> myListIterator = myList.iterator();
  - Map<Integer, String> myMap = new HashMap<Integer, String>();  
Iterator<Integer> keysIterator = myMap.keySet().iterator();  
Iterator<String> valuesIterator = myMap.values().iterator();

What does the Iterable<T> interface look like?

```
public interface Iterable<T> {
 Iterator<T> iterator();
 default void forEach(Consumer<? super T> action){
 //...
 }
 default Spliterator<T> spliterator(){
 //...
 }
}
```

• 2 optional default methods that we are going to ignore for now.

- the iterator() method that, upon implementation, should create and return a new iterator for the collection

(by the language developers)

What does the phrase "syntactic sugar" mean?

- A phrase used to describe a programming language feature that is created for convenience and is easy to use and type.
  - but in reality, behind the easy-to-type code, there is something more complex happening
  - the "syntactic sugar" just hides the details of the more complex process.
- aka an "enhanced for-loop", it is Java's language-level support for the iterator pattern.
  - "syntactic sugar" for iteration
- for-each loops are much less tedious & also easier to code out than while-loops for iterators; basically you (the programmer) can use a for-each loop to work with the elements of a collection, and behind-the-scenes, the compiler uses iterator to translate the actions in the for-each loop!
- No access to the index number of each element.

Are there any disadvantages to for-each loops?

What does a for-each loop look like & how do you use one?

for (T item : collection) {

- declare a local variable that references each item (similar to `i` in `for(int i=0...){...}`)
- the local variable's type must match that of the items in the collection

// do something with each item

}

• reference to the collection, which must be an `Iterable<T>` object

→ Example:

`List<Integer> ages = new ArrayList<>();`

`ages.add(); ...`

(adding several values to the collection)

for (Integer age1 : ages) {

`System.out.println(age1);`

}

for-each loop only works  
if ages is iterable in type  
Integer

printing every Integer item in  
the collection

What is happening behind the scenes of the enhanced for loop?

→ the compiler calls `next()` on the iterator with a while loop:

`Iterator<Integer> iter = ages.iterator();`

`while (iterator.hasNext()) {`

`Integer age1 = iter.next();`

`System.out.println(age1);`

}

→ These 2 while and for-each loops are thus basically equivalent.

## Decorator

What is **decorator**?

→ A design pattern that allows us to "extend" or modify the implementation of an interface without subclassing/inheritance.

→ Instead of subclassing, we modify the implementation class of an interface by relying on an existing instance, & layering different functionalities on top of that instance.

→ Inheritance is not very helpful for programs that want to compose functionality out of different parts

- i.e., if you have a main idea for an object & want to be able to pick & choose different features (aka alternate "types" of the main obj) & combine features to create one specific type of that object

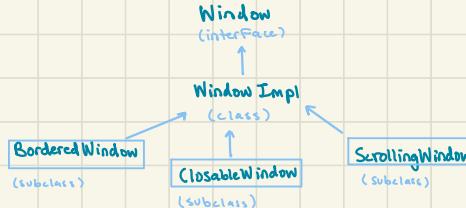
→ Recall! → A class cannot extend from more than 1 parent class

→ Examine the following example

- say you are building a GUI window object. There are many different possible features you may want to add to the window - a border, a scroll bar, an exit/close window feature, etc.

• We can start by defining a **Window** interface for the most basic/simple version of a window, and a **WindowImpl** class that implements it

• Imagine that the "features" are relatively complex to code & therefore warrant their own classes/object types ... but they all start at the same simple baseline, so we can extend each one from **WindowImpl**:



Example scenario where inheritance would not be effective?

(orange notes = referring to window example)

What if we want to have a **Window** that is both **closable** and **bordered**?

→ At subclass only has the power to modify 1 class (its parent), so we either have to:

- a) create a new **BorderedClosableWindow** subclass that extends from the main **WindowImpl**
  - b) have **Bordered** extend from **Closable** (or vice versa) in order to create this combination
- \* but then, what if I want a **closable**, **scrollable**, NOT **bordered** window?

→ These options suck because they require us to create whole new classes for every possible combination of features, which is a TON of repeating code - which literally defeats the whole point of using inheritance in the first place!

→ We also have to "presuppose" all the different combinations in advance rather than on-the-spot

→ Inheritance does not give us the ability to combine functionalities/features of an abstraction... this is where **Decorator** comes in!

Bottom Line/Summary:

## - How Decorator Works, Conceptually -

- In the decorator pattern, we have 2 kinds of classes – base classes & decorator classes – all of which implement the (one) overarching interface for the abstraction.
- the base & decorator classes are decoupled (no inheritance)
- The classes which actually implement the interface (with the most basic version/implementation) of the object
  - Window Impl

What are Base classes?

→ classes that "decorate" an existing instance of the **base class** with additional and/or modified functionality

→ they aren't full implementations (of the interface) themselves, but instead rely on an existing object (the base class object) that already has most of the interfaces required implementations.

- 1) First, it asks for an instance of the interface object (in the constructor)
- 2) Then, it "layers itself" on top of the existing instance by delegating to it for most of its functionality, in order to add on some additional feature
  - as well as potentially modifying some of the instance's functionality, as needed.

• Scrolling Window

• Closable Window

• Bordered Window

→ As many as it wants!

1. take an existing object of the interface/base class
  2. create a new instance of a decorator class and "wrap" our object in its functionality
  3. create a new instance of another decorator class and wrap this same object in its functionality.
- ... and so on. We basically take 1 initial instance & modify it several times.

→ It must provide code implementing every method that is defined in the interface.

→ By **encapsulating an instance of the same interface which it implements** & which already has the required implementations – aka the base class object – & calling its methods.

• this object already does "most of what we want."

→ To implement the methods that the decorator class does not need to modify:

• the decorator class "delegates" the action to the encapsulated base object

by simply calling that object's version of the method & returning the result!

(this will make more sense after viewing the example)

→ To implement the methods that we do want to modify / add functionality to:

• same process, but we add code, either before or after "delegating" (aka the method call), that modifies/ replaces/ adjusts the behavior of the base object's method.

→ The decorator class can also choose to not delegate & just replace the behavior completely, if desired.

What are Decorator classes?

How does a decorator class work?

How many decorator classes can an instance object utilize?

### REASON:

What is the requirement for any class implementing an interface?

Why can a decorator claim to implement the Interface?

## -How Decorator Works, In Code-

What does the decorator class look like?

```
public class BorderedWindow implements Window {
 private Window ogWindow;

 BorderedWindow (Window ogWindow) {
 this.ogWindow = ogWindow;
 }

 @Override
 public String addText (String text) {
 String result = ogWindow.addText (text);
 return result;
 }

 @Override
 public double paintWindow () {
 double result = ogWindow.paintWindow () +
 3 * (ogWindow.getSize ());
 // (basically some behavior that modifies the method)
 return result;
 }
}
```

The constructor takes an instance of the interface as a parameter.

• it encapsulates this instance of a plain window as a field, and will ultimately modify this field's object to have a border, and return that to us.

all of the methods are marked as overridden.

• A method required by the interface but unrelated to the features given by BorderedWindow

• Instead of implementing the method code directly, methods delegate to the encapsulated instance

• A method required by the interface that BorderedWindow wants to modify

• We still call the encapsulated instance's method, but then modify the result before returning it.

How do we create and use a decorator object?

1. create an object of the base class
2. create an object of the decorator class & pass in the base class object as a parameter, effectively "wrapping" the base object in a decorator object.

```
Window baseWindow = new WindowImpl();
Window windowWBorder = new BorderedWindow (baseWindow);
Window BorderedClosableWindow = new ClosableWindow (windowWBorder);
```

What does it mean to "chain decorators"?

- When you take an interface-type object that has already been decorated, & decorate it again by wrapping it in another decorator object
  - like the example code above
- When we chain decorators, we essentially create a linked list of Window objects in memory, because when we actually use the "most decorated" object in Main, it follows the series of objects in order to retrieve info

Main.java :  
 Sout (BorderedClosableWindow.getName());

→ imagine that getName() is unmodified by any decorator, so the actual info lies in the base class object.



What does it mean to unwrap a decorator?

How do we unwrap a decorated object (of "Interface" type)?

- When you want to extract/access the instance of the original object from the decorated one.
- We do this by creating (and calling) an "unwrap" method in the decorator class
  - since this method is added on by a decorator class and isn't a part of the interface, we can only use/call it on objects of type `Decorator`
- Since our reference to the object we want to `unwrap` cannot be of the interface type & must be of the decorator class' type, we must **typecast (downcast)** the object before **unwrapping it**.

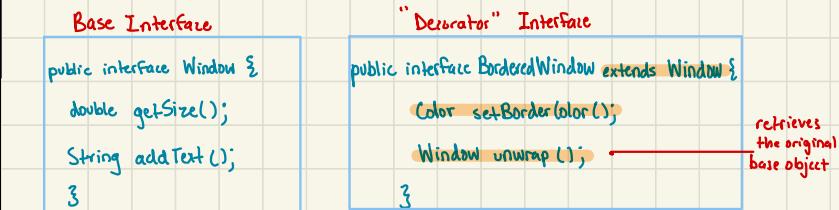
at this point, "closable" is of type `Window` and thus does not have access to the "unwrap()" method

```
Window ogWindow = new WindowImpl();
Window closable = new ClosableWindow(ogWindow);
Window unwrapped = (((ClosableWindow) closable).unwrap());
```

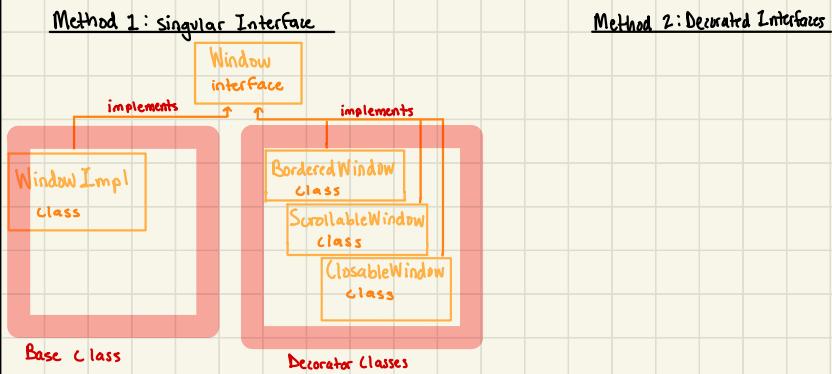
`Window` must be typecast to `ClosableWindow` in order to access `unwrap()`

What is the 2<sup>nd</sup> common way to use the Decorator Pattern?

- Rather than decorating base class implementations of an interface with decorator classes that also implement that same singular interface...
- We can create a **Decorator Interface** that
  - 1) extends a base Interface, and
  - 2) adds (definitions for) extended functionality methods
- And then our "decorator classes" will just be implementations of the decorator interfaces.
- Example:



Comparison of the 2 common Decorator Design uses?



(for Method 1 :)

1. Implement - make a new class that implements the interface
2. Encapsulate - wrap another instance of the interface inside the new class
3. Delegate - Forward (delegate) all methods to the encapsulated instance.
4. Modify - Selectively add or change method functionality as desired.

→ Multiple decorations must be managed by the programmer

- Does order matter?
- Are some decorations incompatible with each other?
- What if the same decoration is added multiple times?
- basically a lot to think

→ No access to encapsulated object's protected fields.

- can only work with the public methods & fields of the base class (as opposed to if we had an inheritance / subclassing relationship).

Summary: What is the Decorator pattern Recipe?

What are the limitations of the decorator pattern?

# Singleton Design Pattern

What are creational design patterns?

→ design patterns that control how to create (and create) a new instance of a class.

How do we usually create new instances of a class?

→ A constructor that populates new instances with states (by filling in the fields).  
→ **Instantiate**: to create an instance of a class using the class constructor and the **new** keyword - which asks for an entirely new instantiation to be initialized in memory.

What are some common creational design patterns?

→ Abstract Factory → **Factory Method**  
→ Builder → Prototype  
→ **Singleton** → **Multiton**

→ **Singleton**, **Multiton**, and **Factory Method** are employed when we want to prevent the use of the constructor.

→ Controls when and where new instances can be created.

→ restricts instantiation of a class to one "single" instance.

→ A class that represents a finite system resource

• like a camera or another system-wide resource

→ A class that is expensive to instantiate, but an instance can be reused.

• a class where it takes a lot of memory, resources, or computational power to create the object instance

→ A class that is used to coordinate different parts of your code

• like a log of logging errors or debugging info - the log is a system-wide resource that we want to be able to add to or send messages to, & have every program in the system be using the same log  
• different parts of your code want to use the same coordinated resource.

→ For a particular class/object, we want to be able to

a) be able to instantiate an object of the class if one has never been created (a one-time action/occurrence)

b) once one has already been created, then anytime we use the class/object, we always only want to get back that same exact existing instance (as opposed to creating new ones).

→ **RECALL:** The way that a constructor is invoked is with the keyword **new**;

• The automatic action behind **new** is to allocate memory & create a new object of that class for the constructor to fill - **new always creates a new object**.

→ By the time that the constructor is called, its too late to prevent instantiation

- there is no way to prevent a new object from being created (other than throwing an exception)

What does Singleton do?

When would it be useful to have a class that only ever has 1 instance?

What is the goal of the Singleton pattern?

Why can't a regular class constructor achieve this goal?

How does Singleton execute this goal?

- Singleton, Multiton, & Factory Method's big idea: make the constructor private.
  - basically prevent the constructor from ever being called in the first place, so that any code from outside the class is prevented from ever creating a new object.
  - & then also provide a diff way for outside code to access & work with the instance.

What are factory design patterns?

What is their general idea?

What is the 'factory method'?

- A general type of pattern that includes Singleton, Multiton, and Factory Method
- To make the constructor private, and then provide a different mechanism by which outside code can create a new instance when needed, & otherwise return the existing instance if it's alr been created.
  - called the 'factory method'
- A static method inside the class that calls the private constructor, and by which we can actually accomplish the task of controlling instantiation.
- When outside code wants to create/access the class object, it calls this static method rather than calling the constructor.
- A method that is associated with the class as a whole, and not with any particular instance
- accessed using the class name (PositionImpl.getSize()) - as opposed to using an instance name like with instance methods (position1.getSize())

What does a generalized factory dp class look like?

```
public class FrontCamera implements Camera {
 private FrontCamera() {
 //constructor code here
 }
 public static FrontCamera create() {
 //code that controls instantiation and
 //returns a FrontCamera object
 }
}
```

Main.java:

```
Camera c1 = FrontCamera.create();
```

AS OPPOSED TO:

~~Camera c1 = new FrontCamera();~~

call the static method when creating a new object.

## -Employing the Singleton DP-

RECALL: What is a static field?

- A Field which contains just 1 value for the whole class. - a global variable
- aka, every object does NOT have their own version / data value for the field.
- Singleton employs a static field which stores the instance of the class object itself.
  - 1) hide the constructor from public use (private)
  - 2) provide a factory method to create or return a singleton instance.
  - 3) Have a static field that stores that instance for the class as a whole.

general template for factory method DPs

Singleton DP specific

What are the 3 steps in creating a class with Singleton DP?

## Singleton class explanation continued:

- The static field is initially empty.
- The factory method in a Singleton class should check if the object has already been instantiated by someone before
  - if not, it then creates the new instance and stores it in the static field.
    - \* this is known as "lazy initialization"—if no one ever asks for the object, we never create it. \*
- Then, everytime a new instance is tried to be created after that, the method simply returns this static field!

Example of a class enforcing  
Singleton?

```
public class FrontCamera implements Camera {
 private static FrontCamera singleton; // Static singleton field
```

```
 private FrontCamera() {
 // constructor code here
 }

 public static FrontCamera create() {
 if (singleton == null) {
 singleton = new FrontCamera();
 }
 return singleton;
 }
}
```

What are the benefits &  
criticisms of Singleton?

### Benefit

- We can call the factory method from anywhere in our code w/o having to coordinate with our other code about whether the instance has been created already.

### Criticisms

- The Singleton DP might be a little 'overkill' — it's essentially just a fancy global variable...  
not all that different than just making (for ex) a `FrontCamera` class that contains nothing in it but a single `FrontCamera` object as a static field.
- We have to know for sure that multiple instances won't ever be needed...  
if not, it's usually best to keep code general enough to support multiple instances.

# Multiton Design Pattern

What is multiton?

Example of an abstraction that would utilize multiton?

Why is it important to only have 1 object for each 'unique characteristic'?

What are the steps in creating a class that enforces Multiton?

- A generalized form of the singleton design pattern.
- instead of a single object that everyone shares, we have a collection of objects where each object is uniquely identifiable by some specific characteristic.
  - for that unique piece of info, we only ever want to have & use 1 same object instance.
- Imagine a Student class where you always get the same object to represent the same student — and every student has a unique PID number
  - there should only ever be one Student object instance created for a given PID number
  - whenever an object to represent a certain PID is needed/called/attempted to be created by outside code, the same object should always be returned.
- This is especially important if the object is mutable (i.e., if users are able to change aspects of the object (name, address, pronouns, etc.))
  - we'd want to make sure that every other part of the system that is using the same student instance that is being mutated, is able to see those changes.

steps:

- 1) Store a collection of instances as a private, static field; this is our "directory" — as opposed to storing one instance (one "singleton") in the Singleton DP
  - aka a HashMap (which we initially instantiate as empty) that is a mapping between each object & its uniquely identifying characteristic (uic)
- 2) Create a private constructor.
- 3) Create the static factory method (our factory for creating new object instances);
  - the method accepts all of the same parameters that the constructor would have (since it kind of functions as the 'constructor' used by outside code)
  - in the method, provide all of the information necessary to create a new instance of the class object — possibly including the uic as well.
    - ... a.k.a. invoke the constructor.
- 4) (In the FM): first search the collection directory to see whether an object for the provided uic has already been created
  - \* if not, instantiate (and return) a new object & add it to the collection
  - \* if yes, then simply return the value already mapped to the uic in the collection

Example of a class  
enforcing Multiton ?

```
public class Student {
 private static Map<Integer, Student> directory = new HashMap<>();

 private Student (int pid, String first, String last) {
 //constructor code here }
 public static Student getStudent (int pid, String first, String last) {
 if (!(directory.containsKey(pid))) {
 directory.put (pid, new Student (pid, first, last));
 }
 return directory.get (pid);
 }
}
```

the factory method

for the same pid, we always get the same object

## Factory Method Design Pattern

What is the Factory Method

DP?

→ For an interface or parent class that has a number of implementing subclasses and where we want to force the use of a single subclass, a Factory Method exists to dynamically choose which subclass to use & instantiate an object with.

- the decision of which subclass to use is usually subject to some complex logic that we don't want the user of the abstraction to have to know or care about.
- User doesn't even have to know or care about what subclasses there are — they just call the FM in the parent class & are returned with an object of the appropriate subclass type — but the reference type is the parent class.

→ preventing creating of objects (of the parent class type) directly, & instead making the class in charge of deciding which subtype object gets made.

→ Unlike Singleton & Multiton, the static 'factory method' that instantiates objects is not in the same class as the class where we are wanting to prevent instantiation.

→ for all of the subclasses whose direct instantiation we are trying to prevent, our factory method is written just once, in the parent class.

→ This method takes in all the information needed to make a decision, & then contains code that figures out which subclass should be used (the "complex logic")

→ It then returns a new instance of the appropriate subclass type.

→ They can't.

→ Instead, we declare the constructors of the subclasses as protected.

In what way does Factory Method "control instantiation"?

How is Factory Method different from Singleton & Multiton?

What does the static fm method of a Factory Method DP do?

But how can our subclass constructors be private if the fm needs to invoke them from inside the parent class?

Example of an abstraction that employs Factory Method?

→ Consider a Notification object/class that has several subclasses of specific types of notifications — TextNotification, EmailNotification, PushNotification

→ Goal: user should be able to create a new Notification object that corresponds to a given Student object, & the class should return the correct type of notification for that specific user.

(For ex, this could be based on a getNotificationPreference method in the Student class)

What would the parent class look like in this example?

notice that the static fm  
is actually inside the  
parent class' constructor

The parameter provides the info  
needed to choose a subclass

the factory  
method

```
class TextNotification extends Notification
class EmailNotification extends Notification
class PushNotification extends Notification

public class Notification {
 public enum Type { TEXT, EMAIL, PUSH }

 public static Notification create (Student s) {
 Type test = s.getNotiPreference();
 if (test == Type.TEXT) {
 return new TextNotification();
 } else if (type == Type.EMAIL) {
 // and so on for each type
 } ...
 }
}
```

What would an outside class using an FM class look like in this example?

Main.java

```
Student stu = Student.getStudent (123, "Arif", "Kumar");
Notification n = Notification.create (s);
```

↑  
notice that while the object  
returned will be of a specific  
Subclass, the reference type is  
always of the parent class.

Use the Factory method to  
dynamically instantiate  
the correct subclass.

What is the advantage of using  
Factory Method?

- It separates the encapsulation of the object from the rest of your program
- It associates any logic that is needed to choose a subclass all into one spot (the factory method)
- Makes it super easy to use this class & retrieve a value (just look at Main.java example above!) — outside users don't need to know anything about how the decision gets made.

## - Recap: Singleton, Multiton, Factory Method -

→ All three are creational design patterns

### Singleton

- private constructor
- static private "singleton" instance
- static "fm" method for lazy creation/receiving the Singleton instance.

### Multiton

- private constructor.
- static private collection of instances
- static factory method for creation/retrieving instances corresponding to a uniquely identifying characteristic.

### "Factory" Method

- static method in parent class for instantiating an object for a particular context
- "protected" constructors (in subclasses) so that the static method can access them
- Typically used for dynamic subclass binding, for deciding which subclass should be used.

# 'Strategy' Design Pattern

- What type of DP is Strategy ? → A Behavioral Pattern - patterns involved in helping define the behavior of one object/program with respect to something else.
- What is the motivation behind Strategy DP ?
- Imagine that we are writing an algorithm that is completely definable & written except for one critical missing part
    - this "missing part" may perform an action or return some object that we need and that we know how to/have already written code describing how to utilize - we just don't know how to obtain this missing piece.
    - or maybe there are multiple possibilities for what the missing piece could be, and we (as the algorithm-writers) don't know which one will be the most appropriate for a given instantiation of the alg.
  - Strategy allows us to go ahead & implement that general algorithm without the missing piece , and then inject the appropriate missing "strategy" at the time that the alg needs it.
  - RECALL : Dependency Injection ; Inversion of Control ;
  - When the algorithm is running & it gets to that critical step , it then gives the control over to the 'strategy object' to do whatever it needs to do & then return the answer to the algorithm so that it can continue running .
  - Choosing a sorting algorithm
    - A general program that, at some point, needs to perform a sort of items . Rather than choosing & implementing one sorting algorithm for all circumstances, we can allow the specific sort alg ( BubbleSort , Inception Sort , Selection Sort , Hemp Sort , etc. ) to be chosen at the time that the program is being used
    - Because we might want to use a different sorting alg depending on different circumstances that aren't known until the time of our program's use ;
      - for ex - the size of the collection (bc have to consider the time & space efficiency of the sort )
  - Determining order of some items
    - A sorting algorithm that, at some point, needs to compare 2 items to decide which should come first .
    - Instead of writing the sorting alg for the specific characteristics of the object (like alphabetical sorting, for ex ), we can allow the user to inject their own comparison object that will determine the correct order
    - Our alg doesn't need to know why or how , just needs to know which object comes first.

## → How to output results

- An algorithm that has created results & produced some output... how that output should be communicated doesn't have to be known by the alg in advance & instead can be provided to it at the time that it actually needs to produce its results
- Alg can then "hand over control" to some strategy object (given by the user), which will then produce the correct type of output
  - For ex: a summary on the command line v.s. insertion into a log v.s. written to the disk v.s. ... etc.

What is the general structure of Strategy?

## → 3 Key components:

- 1) An interface that represents "the strategy" — the interchangeable object that performs some action & that will be injected into some larger piece of code when it is needed.
- 2) The critical method signature defined by the interface
  - Usually, strategy interfaces only define 1 single method — but not always (there could be multiple)
  - This is the method that the outside code will call — it represents some alg to be performed.
- 3) Several implementing classes — called "Strategy objects" — that each offer a different way to perform the algorithm / implement the critical method.

What is an example of Strategy DP?

→ The Comparator Interface, which is built into Java Core. :

② The interface represents a strategy for comparing 2 objects of type T;  
it is a generic interface (RECALL: parametric polymorphism)

```
public interface Comparator<T> {
 int compare(T o1, T o2);
}
```

③ the interface's only method (the "critical method"), which takes in 2 objects of Type T & returns an int val that represents the order;

val < 0 : o1 before o2

val = 0 : o1 is equal to o2

val > 0 : o1 after o2

What would the "strategy objects" look like in this example?

→ First, imagine a Shape object that has a double getPerimeter() & a double getArea() method.

→ Here are 2 strategy objects that implement Comparator<Shape> in different ways:

```
1 public class SmallestPerimeterFirst implements Comparator<Shape> {
 public int compare(Shape o1, Shape o2) {
 return (int) (o1.getPerimeter() - o2.getPerimeter());
 }
}
```

So how would we then use this strategy?

2

```
public class LargestAreaFirst implements Comparator<Shape> {
 public int compare(Shape o1, Shape o2) {
 return int(o2.getArea() - o1.getArea());
 }
}
```

→ Let's say we have created some *List* of *Shapes* that we now want to sort;

```
List<Shape> shapes = new ArrayList<>();
shapes.add(shape1);
shapes.add(shape2);
shapes.add(shape3);
shapes.sort();
```

→ the *List*<> interface defines/includes a built-in *sort* method which sorts the objects according to a specified order — this is the "general algorithm" or the "bigger program"

→ the algorithm knows everything about how to sort items but is missing that last critical piece, which is how to go about comparing 2 objects... that is what we have to inject (RECALL: Dependency Injection)

This is where we pass in a *Comparator*<> object to tell the *sort()* method how to compare the objects — our strategy injection.

→ We can inject the desired strategy into the *sort()* algorithm by creating a new strategy object:

```
shapes.sort(new SmallestPerimeterFirst());
```

or

```
shapes.sort(new LargestAreaFirst());
```

## Observer D.P.

What situation does **Observer** address?

- Situations where you need to write code that responds to an action or an event occurring.
- Something is happening inside one object, to which another object must respond.
- For example, a **Button** object would be a good subject - the observer objects would represent & execute whatever action should occur when a user clicks the button.
- **Observer DP is often at the heart of asynchronous systems/software.**

What are asynchronous systems?

- programs that are built to respond to events/circumstances where it doesn't know when that event is going to happen.

→ For ex, **User interfaces**

→ User Interfaces

- Events may be caused, for ex, by hardware or user input

• web programming with JavaScript

• Graphical User Interfaces (GUIs); button, mouse, scroll, text, pop-up, etc... all user

actions that require some coded response action)

→ As a Building Block for other DPs

• Model-View

• Model-View-Controller

- Code/program is organized into 1 **subject object**, & 1 or more **observer objects**
- Each observer object defines a special method of code that they want to execute in response to some specific "event" occurring with the subject object - the "**update method**"
- The subject object is then responsible for notifying all observers that the action has occurred.

What is the design pattern?

→ A state change occurring inside the subject object

→ For example, could be induced by :

- User interaction with an on-screen UI component (a user pressing a button)
- Hardware (sensors, buttons, etc.)
- Changing a field value with a setter method.

→ It encapsulates a list of all active observer objects ... as well as 2 (public) methods that allow observer objects to add or remove themselves from the list.

1. Observer objects register with the subject object (added to list)
2. An event occurs inside the subject object
3. The subject object notifies all the observers one at a time by calling the "update method" of each.
4. The observers then react to the event.
5. Observers can "deregister" to stop observing.

How does the subject object know who its observers are?

What is the execution sequence for a program using **Observer**?

Key terms: what is ...

a **subject object** (subj0)?

→ An object that causes an event to happen

an **observer object** (obs0)?

→ An object interested in the event

→ Even if they are different objects (diff implementations), All obs0s must be implementing the same common interface

- because the Subj0 will register them as [interface] type objects & will call the same "update" method for each obs0

registering?

→ The act of connecting an observer to a particular subj0.

deregistering?

→ The act of no longer observing an event.

What does a basic subject class

look like?

```
public class Subject {
```

```
 private List<Observer> observers;
```

the type of the list is the interface  
that the Subj0 expects all of its  
observers to implement.

Encapsulates a list of observer  
objects

```
 public void addObserver (Observer o) {
```

```
 observers.add (o);
```

```
}
```

allows obs0s to be registered  
or deregistered from the list

```
 public void removeObserver (Observer o) {
```

```
 observers.remove (o);
```

```
}
```

```
 private void notifyObservers () {
```

```
 for (Observer o : observers) {
```

```
 o.update ();
```

```
 }
```

a method called from within the  
Subject (hence private) that calls  
update() on all observers, allowing  
them to respond.

- most basic form: a for-loop

What does a basic observer  
interface look like?

→ In the most basic case, just contains a single method - the "update  
method" - that the subject needs in order to notify the observers;

```
public interface Observer {
```

```
 void update();
```

```
}
```

What does a basic observer impl class look like?

Encapsulates the subject being followed

```
public class ObsImpl implements Observer {
 private Subject subject1;

 public ObsImpl(Subject s) {
 subject1 = s;
 subject1.addObserver(this);
 }

 public void update() {
 //For example:
 System.out.println("You scored!");
 }
}
```

Adds itself as an observer of the subject

When an event occurs, this code is executed

What are the limitations of this basic version of the Observer DP?

1. If an observer is registered to /wants to be notified by more than one subject object, it has no way to know which subject is notifying it
  - update() method will get called... but by which subject?
2. The obs0 doesn't know/receive any information characterizing how the subj0 has changed, just that a change has occurred — no event context.

What modifications can we make to solve these?

Limitation ①

→ have the update() method take a subj0 as a parameter!

Improved subject class:

```
... private void notifyObservers() {
 for (Observer o : observers) {
 o.update(this);
 }
}
```

Improved object class:

```
public class ObsImpl implements Observer {
 public void update(Subject s) {
 System.out.println("You scored!");
 }
}
```

• No longer need to encapsulate the Subj0 inside the obs0 class

• Object can code different reactions based on what subj. called the method.

→ have the update() method take in event information as a parameter too!

→ This is commonly done by creating an event object class that encapsulates all of the useful contextual info describing what occurred. Then:

- 1) have the subject class create a new instance of the EventObject everytime the event occurs
- 2) pass this event object as a parameter into the notifyObservers method
- 3) have notifyObservers pass this object into the update method for all observers.

Limitation ②

how can we pass in event info?

Example of the improved Subject & object classes?

New class defining an event:

```
public interface Event {
 //for example:
 String getType();
 String getTeam();}
```

Further Improved observer class:

```
public class ObsImpl implements Observer {
 public void update(Subjects , Event e) {
 System.out.println("you scored!"); }
 }
```

The update() method now takes in 2 parameters;

1. which object produced the event

2. what actually happened

Further Improved Subject class:

```
public class Subject {
```

...

```
 public void actionOccurring {
```

//for example:

```
Event e ;
```

```
if (...) {
```

```
 e = new EventImpl (xx,yy) ;
} else if (...) {
 e = new EventImpl (aa,bb) ;
```

```
 notifyObservers (e);
```

3

```
private void notifyObservers () {
```

```
 for (Observer o : observers) {
```

```
 o.update (this, e);
```

333

the method where the event takes place

Whenever the event occurs, a new Event object is created

The action method then calls notifyObservers - this time passing in the appropriate Event object

## — Alternate ways to support Observer programs with multiple events —

### 1. Separate observer interfaces

Motivation behind this method:

→ Currently, we only have 1 EventImpl class that encapsulates the name & etc. of a given event, and no matter what the event actually is, all observers get notified of it (& thus run their respective programs)

- but what if only a few observers actually care about the event? For ex,

a UNCFan doesn't care about or need to respond to an event where Duke scored points (which is why the UNCFan contains an if statement that seeks to deduce the details of the event that occurred);

```
if (e.getName == "UNC") {
 SOUT ("yay!");
}
```

- Want to avoid having to run the UNCFan update() method entirely

→ Instead, the subj0 might want to support separate events that all fire independently.

→ Separate observer interfaces for each type of event — e.g., a different obs0 object/class for each kind of event.

- obs0 classes can pick & choose which event interfaces they implement ("register with"), so that they are only notified at a time when they'd need to perform an action.

Pros and Cons:

- ★ Allows the observer to only process what it wants

✗ requires subject to provide separate registration & notification methods for each event — more tedious in the Subj0 code.

Example:

→ public interface UNCScoreObs { ... }

interfaces for each type of event

public interface DukeScoreObs { ... }

→ public class UNCscoreObsImpl implements UNCscoreObs {

public void update() {

SOUT ("yay!");  
 }

obs0 class... now it  
doesn't need the if  
statements!

(continued next page)

```

→ public class Subject {
 private List<UNCObserver>; • Subj0 must encapsulate
 private List<DukeObserver>; separate lists for each
 public void actionOccurring { observer
 //for example:
 Event e;
 if (...) {
 e = new EventImpl(xx, yy);
 notify UNCObservers();
 } else if (...) {
 e = new EventImpl(aa, bb);
 notify DukeObservers();
 }
 }
 private void notify UNCObservers() {
 ...
 }
 private void notify DukeObservers() {
 ...
 }
 public void addUNCObserver(UNCObserver o){...}
 public void addDukeObserver(DukeObserver o){...}
 (and so on...) 33
}

```

• subj0 decides when to notify which observers — aka, only when the action that occurred is relevant to them

• subj0 needs to define update(), add(), and remove() methods for each observer type

• could get tedious if there are many

## 2. Single observer interface with multiple "update()" methods

Motivation:

→ Similar to that of alternate method 1 — a program that has several obsr types and many different possible events would benefit from some way to organize & streamline the observer notification & response process.

→ A single interface definition (that all obsr implement) that defines separate update() methods for each kind of event;

updateThreePointer();      updateDukeScored();      updateUNCScored();

→ the subj0 chooses when to call which method (based on what happened), but this notification does get sent out to all observers (unlike alternate strategy #1)

\* Observer must provide/implement all the methods required by the interface, even the ones that are assoc. w/ events they don't care about  
— the method can just be empty (since its **void**)

\* reasonable approach if most observers will want to process most event types.

Pros & Cons:

# Functional Programming

What is functional programming?  
(FP)

- The Functional programming approach allows functions to be their own code "objects" to be executed, passed around as arguments, modified, created, stored as variables etc.
- Essentially, "functions" are the primary thing (the "first class citizen") being instantiated & utilized, rather than entire objects.
  - This approach is a useful pattern for coding many real-world scenarios.
- emphasizing the function rather than the data
- In OOP, the primary organization of our code is around the data type
  - our program's purpose is to create particular kinds of objects (which are basically collections of data), & to have them interact with each other.
- In contrast, FP focuses on functions themselves as the things that we create, modify, and work with
  - functions are kind of "objects" in and of themselves.
- A block of code that whose purpose is to execute a particular action (often specific to/utilizing data that is provided to it).
  - RECALL: Instance methods... "functions" are sort of the same, in concept/purpose.

Comparison of object-oriented  
versus functional programming?

| Object-Oriented                   | Functional                                         |
|-----------------------------------|----------------------------------------------------|
| primary unit of code organization | → classes /objects                                 |
| data storage & execution          | → mutable fields grouped into classes              |
| code execution                    | → methods that mutate their data (w/ side effects) |

→ Observer object classes in the Observer Design Pattern!;

- the whole point of the entire classes impl an Obsr interface is simply to define a single update() method (which is basically a function/algorith)
- they have no fields or constructors or anything

```
public interface Observer {
 public void update(Subject s, Event e);
}
```

How do coding languages support functional programming?

- In a more FP-oriented programming lang (like Javascript), we would be able to execute this DP w/o needing those obsr classes—the subj could simply call the functions, because the functions themselves can be passed in as an argument, stored on a list, & then called again later.
- Most other languages (besides Java) have built-in features that allow functions to be used more flexibly—for ex, being able to store a function as a local variable.
- Java eventually began adding some limited support for FP.

Before FP-support in Java, what was the traditional way to execute FP-based projects?

- To use a "function" in a Main class, you would have to do 3 separate things:
  1. Create a .java Interface file that defines a method (the "function")
  2. Creating a .java Class file that implements that interface & defines that "function object" in some implementation-specific way (think UNCFan versus DukeFan).
  3. Immediately create an instance of that class (in your outer/current file).
- a.k.a. the same approach we have been learning & doing for all the assignments in class so far.
- for ex, the **Observer DP** strategy described in prr. chapter of notes.
- The "programming to an interface" method was created for (and is useful for) Object-Oriented Programming
- Coding FP with these OOP features is tedious & feels like too much coding
  - anytime we want to incorporate an interface that only defines one method, we have to create a whole new .java file implementing it!
  - What if that specific implementation only needed to be used once in your program? Creating a whole new class is excessive.

What FP-support did Java eventually add?

- Two new language constructs that, for any interface, allow us to carry out steps 2 and 3 (prv.page) all in one expression!
  - anonymous classes
  - lambda expressions
- They effectively result in a reference to an object that implements some interface.
- You still have to have an interface to define the function we are providing an impl for; Java is still an OOP language - these 2 features are just syntactic sugar to make it look more like an FP one.

### - Anonymous Classes -

What is an anonymous class?

- A class that is:
  - defined/created inside another class rather than having its own .java file (an "inner class")
  - doesn't have its own name
  - A class for which only a single object is created.
- Allows us to make our code more concise because we can declare & instantiate a class at the same time.
- A way to quickly make an instance of an existing interface w/o creating a new file
- Say we have this existing interface:

```
public interface Fan {
 void update(Game g);
}
```

How do you create a new anonymous class?

→ Rather than creating a new `UNCFan` class which implements `Fan`, we can create a new instance of the interface object (in our main/outer code file), & then define the method(s) of the interface right then & there!

Example?

`Main.java:`

`Fan tarheel = new Fan() {`

`@Override`

`public void update(Game g) {`

`if(g.whoIsWinning().equals("UNC")) {`

`System.out.println("Go Heels!");}`

`} ;`

• creating a new instance of the interface type,

which we would otherwise NEVER be able to do (since all it contains is method signatures)

• Right where we are declaring the instance, we are also creating the anon class body - hence, brackets after the new instantiation.

• the interface method is defined directly in the anonymous class body

→ every time we create a new `Fan`, we are creating a different anonymous class (even

if we were providing the exact same code in the body), and creating exactly one instance of this class

→ For purposes of identifying, these anonymous classes do have "names", which are random & are chosen by Java - but all of this is hidden from us b/c we don't have to care about that.

## - Lambda Expressions -

What are lambda expressions?

→ A type of anonymous class for interfaces that define only 1 single method.

\* like the `Observer` ... interface!

`Main.java:`

`Fan tarheel = (Game g) -> {`

`if(g.whoIsWinning().equals("UNC")) {`

`System.out.println("Go Heels!");}`

`} ;`

• Instead of using the `new` keyword, the method's parameter list goes in parentheses, followed by an "arrow" →

→ This parameter list must match the one of the interface's method.

Since the interface only has 1 method, we don't have to specify the name & instead can just go ahead & define the method body.

How is this Functional Programming?

Similar to

→ this structure, although it still technically is defining a new class "object", makes the instantiated object look more like a function

\* the `tarheel` variable looks like it is equal to the function defined below, but in reality, Java has created an object class (of some random name) that implements `Fan`, and assigned `tarheel` to be equal to an instance of that class - so, still an object.

# Midterm 2 Study Guide



key info  
subheadings  
secondary key info  
key terms

## Unit 6: Error Handling

→ Exceptions: unexpected/unusual situation which arises during execution of a program... can (& should) be anticipated & dealt w/ by the program whenever possible.

### → Early error-handling strategies

1. Global error codes: a global variable where we store an int representing an "error code" whenever something goes wrong.

- declare a public static int @ top of a class

- Anytime code does smthn where an error could occur, we have to check the variable to see if it is still 0 or has changed

### → DRAWBACKS TO EARLY METHODS:

- \* Reliant on documentation (made by the programmer) explaining what each error code/return val means — this documentation needs to be well understood by others using the program

- \* (global error codes) if 2<sup>nd</sup> error occurs while 1<sup>st</sup> is being handled, there is nowhere to store the code

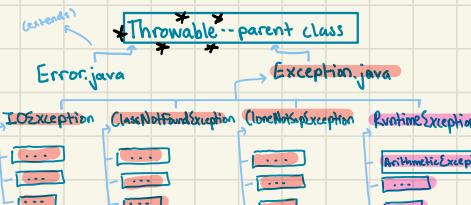
## Unit 7: Exceptions

→ PRO: Safer than old methods; can ensure that any code is executed that needs to be

→ Exception handling: formal method for detecting & responding

### → Exception handling in Java: Exception objects

- objects for each specific type of exception, that encapsulate details about the error that occurred — Java provides many built-in exception classes
- classified with inheritance:



→ "Error" represents externally caused, unrecoverable problems that generally shouldn't be caught/handled

### → Catching an Exception

(i.e., in the Main method...)

→ the "handling" aspect — providing the code to handle a thrown exception.

\* if program fully unwinds w/o error being handled, the program dies.

→ try-blocks: the block of code where we write the code/call the method that has possibility of an exception

→ catch-blocks: the block of code which contains the actual code handling the exception (how the prog responds to a given throw)

- usually multiple catch blocks, each one corresponding to a different type (class) of exception.

4. program jumps through the catch-blocks, looking for the (first) one that defines the same Exception class type (or a parent class of) the thrown exception.

5. executes the code inside the catch-block (ONLY the 1<sup>st</sup> "match" — doesn't look any further)

6.\* if no matching catch-block is found, program returns to "unwinding the stack" & repeating the process w/ the next method on the call frame.

## Unit 7: Exceptions ctd.

→ finally block : placed at the end of the sequence of catch-blocks and contains code that needs to execute no matter what —

- whether or not an exception was thrown
- whether or not it was handled by a catch-block

→ be able to explain the following code execution if `methodBL()`

throws an exception:

```
try {
 methodA();
 methodB();
 methodC();
}
catch (Runtimeexception e) {
 (...)
}
catch (IllegalStateException e) {
 (...)
}
```

ANSWERING DEVICE:

unchecked = Runtime (& throwable & Error) ...

unnecessary to "catch or specify"

checked = everything else... must "catch or specify"

## Unit 7: Checked vs Unchecked Exceptions

### Unchecked Exceptions

(on previous page)

→ RuntimeException & all of its descendants ; the Error class ; and the Throwable class

→ errors caused internally within the program (ex: logic errors) that really "never should have happened" — eg programmer's fault

→ Should only throw exceptions if we know how to handle the situation. May or may not need to address them in our code

→ not subject to the "catch or specify" rule

→ exception is thrown inside the method, but caught in the file where the method is being called.

```
(methodB() { if(x == 1) { throw new RuntimeException(); } })
```

```
(main { try { methodB(); } catch (RuntimeException e) { ... } })
```

### Checked Exceptions

(on previous page)

→ All other Exception subclasses (as well as Exception itself)

→ Responding to errors caused by factors outside the program's control. Our program is responsible for always responding to these

→ Subject to the "catch or specify" rule.

→ exception must be caught (or specified) inside the method itself (not just the file where it is being called.)

→ "catch or specify" rule: if a method contains code that might throw a checked exception, then the method must also EITHER:

A catch the exception internally (with try- & catch-blocks)

• do this iff the current method is the correct place to handle the error (and we know how to deal with it)

OR

B specify in the method signature that the checked exception might

be thrown by the method:

```
public int methodC () throws FileNotFoundException {
```

(best practice rule #3)

• do this if the error needs to be dealt w/ at a higher level

• basically "instructs" the error to bubble up

### "catch or specify" errors

• by specifying an exception in a method, we're basically "putting off" handling it

• we still have to catch it somewhere. 2 options:

1. catch the exception in the Main method by calling it inside a try-block

2. force the exception to continue "building up" by having the Main method

ALSO specify the exception (in its method signature);

```
public static void main (String[] args) throws FileNotFoundException { methodC(); }
```

## Unit T: Compile vs. Runtime Errors

### Compile-time

(vs)

### Run-time

- caught before you run your code
- Syntax errors & etc.
- indicate that something is incomplete about our program
- compiler can't warn us; found upon execution
- All exceptions objects
- Indicate that something is wrong with the logic of our program

## Unit 8: JUnit

→ 4 levels/stages of professional software testing:

### Unit testing

- Testing methods & classes in isolation
- done during development

### Integration Testing

- whether new code/classes work w/ the rest of the program
- during development

### System Testing

- Testing entire system as a whole
- after development

### Acceptance Testing

- "business aspect"; whether the program is meeting consumer needs
- occurs before release

→ Key terms & definitions

### JUnit

- a Java library/framework to help us write unit tests
- statements abt what should be true at some point in the test — they either run smoothly or raise an exception.
- These methods are static.
- `assertTrue(condition)`, `assertFalse`, `assertNull(object)`
- compiler directive to mark which methods inside a test class are unit tests (& not, like, helper methods)
- separate class where we write the unit test methods
- Conventionally: separate test class for each class of the program.

### Test Methods!

- non-static (instance) methods which each test a single method, field, or constructor of a given program class
- return type: void (usually)
- Anatomy of a test method:

1. Create an instance of the program class
2. Use some methods to change that instance's internal state or some other action
3. Use JUnit Assertions to verify that the pc methods return the correct values.

### @Test

```
public void testName() {
```

```
 Product prod = new ProductImpl("shoes", 12.70);
 assertEquals("shoes", prod.getName());
```

### assertEquals versus assertSame

- `assertEquals(expected, actual)`: uses `equals()` method; checking CONTENT equality

- `assertSame(expected, actual)`: uses `==` operator; checking if they are the same object in memory

PNEUMONIC DEVICE:  
`assertEquals` → using `equals()` → The size of my shirt equals the size of your shirt

`assertSame` → My shirt and your shirt are NOT the same shirt

- 3) High test coverage: write many unit tests in order to cover a variety of expected AND edge cases
  - \* ideally, All lines of code in the program class should be executed/used by the test class & methods.

→ When does a unit test fail?

- if an exception is thrown AND is uncaught! if its caught, test will still pass
- when assert methods don't return the expected value, they throw exceptions (this is built into the JUnit assertion method library)

### Best practices in Writing Unit tests

- 1) Isolate unit tests as much as possible: each test aims to test one aspect of a class — although its ok to call multiple pc methods if necessary, bc its hard to test things completely in isolation
- 2) Use more specific assertions when possible because they describe the situation more fully (espec if test fails & we want to know why)
  - for ex, choose `assertEquals` over `assertTrue` when possible

## Unit 9: Iterator

### Iterator Design Pattern - key points

- purpose: be able to access the elements of a collection in some particular sequence WITHOUT "exposing its underlying representation".
  - \* w/o needing to know any details abt the collection (what it is, size, how its being stored, etc.)
- user should just be able to call `(iterator obj name).next()` & get the next object in the collection of data they have provided.
- keeps track of where we are in the collection
- an iterator object: for a given collection, it is a class that encapsulates the details of how to loop through it.
- the iterator pattern/object assumes that the collection will not be modified while the iterator is being actively used.

### Situations where Iterator is useful

1. huge collections (1M+elements, for ex), data too big to store in memory (like in an array)
2. generative collections; sorting through a collection (with no finite size) that creates items on demand

### Iterator <T> versus Iterable <T>

→ both are interfaces provided by Java in support of the DP

#### Iterator <T>

- interface defining 2 primary methods – boolean `hasNext()` and `T next()` that all Iterator object classes must implement.
- this interface is important to us when we are creating a new specific iterator object (like `Alphabetizer`)

#### Iterable <T>

- interface representing a class (usually one representing a collection) that is capable of creating and returning an Iterator object for its elements
- only defines 1 required method: `Iterator<T> iterator()`
- any class can implement `Iterable<T>` so long as they provide that method.
- All of Java's collection classes implement `Iterable<T>` (`Map, List, Set, etc.`)

### For-each loops

→ Java's language-level support ("syntactic sugar") for the iterator pattern.

→ `for (objectType obj : collectionName) {  
 ...  
}`

→ can only be used for objects that implement `Iterable<T>`

→ behind the scenes, compiler uses the collection's `Iterator` object to translate the actions in the loop

## Unit 10: Decorator

→ allows us to "extend"/modify the implementation of an interface by relying on an existing "base" instance & layering diff functionalities on top of it.

### How it works - 3 components

- 1) an interface for the object in general
- 2) A "base class" implementation representing the most basic version of the abstraction object
- 3) Several "decorator classes" implementing the same interface. They encapsulate an instance of the base class (which they take as a parameter of their constructor)

### Why/when is Decorator more efficient than Inheritance?

- One class cannot extend from more than 1 parent class; impossible to add more than one "decoration" or pick & choose different features
- After construction of an object, we can't change the underlying functionality/ data type.

→ Decorator is better if we want to "compose functionality out of different parts"

### Decorator Classes

- They want to "add on" functionality via implementing some (not all) of the interface's methods in a diff way than the base class did
  - previously we would achieve this by creating subclasses of the base class and using `@Override` to rewrite some of the methods
- Instead, the D.C.s implement the interface directly, & for methods that they don't need to modify, they simply call the encapsulated base class' version of the method & return the result.

```
DecoratedItem implements Item {
 private baseItem base;
 public String getName() {
 return base.getName();
 }
}
```

this is called "delegating" (to the base object)

→ For the methods they do want to modify: delegate but add code before or after that adjusts the behavior.

## Unit 10: Decorator (ctd.)

- Decorator decouples the base from the decorator classes (a.k.a. no inheritance)
- When we chain decorators (taking a decorated object as the constructor param. of a another decorator object in order to layer several decorations onto one initial instance), we are basically creating a linked list of the interface objects in memory;
  - Think about it: When Decorator2 takes in a Decorator1 obj & delegates to its methods ... Decorator2 obj then delegates to the base class' methods.

## Unit 11: Singleton & Multiton

- creational DPs controlling instantiation of an object via a private constructor & a static (class-associated) create() method that users will call instead of constructor, when wanting to create a new object.
- the static method - called the "factory method" checks to see if a new instance should be created - this is how the class "controls instantiation"
  - if yes, invokes constructor, creates instance & returns it to user
  - if no, returns an existing object to the user
- the static method takes all the same parameters as the constructor, since it basically functions as the "constructor" for outside users.

### Singleton

- restricting instantiation to one single instance - there should only ever be one existing instance of the class, anywhere, ever
  - Ex: a FrontCamera object
- private, static field (which is initially empty) of the class object that stores the 1 instance (the "singleton") for the entire class

```
private static FrontCamera();
```
- create() / "factory" method checks if obj has already been created (aka if the private field is empty)... if not, creates a new instance right then & there & stores it in the private field
  - Then, returns the private field object (which was either just now or previously created).
  - "lazy initialization" - if no one ever asks for the object, it never gets created.

### Multiton

- every object instance is associated with some uniquely identifying characteristic (like an ID number, for ex)
  - Not like we are assigning a random num to each instance - the abstraction naturally/legitimately already embodies the idea of a U.I.C. for each object (which is why it wants to use Multiton DP in the first place).
- restricting instantiation to no more than one single instance for a given U.I.C.
  - Ex: a Student object where each student has a unique PID
- private static field (initially empty) of a collection (like a HashMap) of instances & their respective U.I.C.s for the entire class

```
private static Map<Integer, Student>
```
- create() method searches the private collection to see if an instance has already been created for the provided U.I.C... if so, returns that instance. If not, creates new instance, adds it to the collection, & returns it.

## Unit 11: Factory Method

→ similar to Singleton & Multiton: creational DP, preventing use of constructor... all three of these are part of a general category of DPs called "factory design patterns"

→ difference from Single/Multiton:

- (a) not restricting the am. of instances that can be created, and
- (b) the `create()` method isn't in the same class as the one where we are trying to control instantiation.

→ purpose: "dynamic subclass binding": dynamically choosing which subclass to use to create a new object with.

### Components:

- 1) a parent class defining some object - like a `Shirt`
- 2) several subclasses defining specific types - `RedShirt`, `BlueShirt`, etc.
  - these subclasses have **protected constructors** (only accessible within their class file by their parent classes)
- 3) a public static "factory"/`create` method that is in the parent class

→ The `create()` / "factory method"

- takes in whatever info needed to make a decision & then contains code that uses some logic/process to decide which subclass type should be used
- returns a new instance of the appropriate subclass (it can invoke the constructors of the subclasses since they are **protected**, not private.)

## Unit 12: Observer

→ purpose: situations where something is happening inside one object, to which another object wants to respond

- user interfaces (like GUIs)... responding to a button being pressed, a mouse click, etc.
- event-driven programming, where events are caused by things like hardware or user input
- as a building block for the Model-View & Model-View-Controller DPs

→ What defines an event? : a state change occurring inside the `subj0`; could be anything from user interaction with a UI component, to simply a field value being changed by a setter method.

### The Subject Object Class

- **Subject**: An object that causes an event to happen.
- **private list** field containing list of all of its active observer objects
- **public addObserver** & **removeObserver** methods which take an `obs0` as a parameter, for observers to register or deregister from the subject — adds or removes `obs0` from its private list.
  - `obs0` calls `addObserver` in its constructor
- **private notifyObservers()** method that calls the `update()` method of all observers in its list... `subj0` calls its `notify` method whenever the event occurs. It's sort of a helper method for the `subj0`

### The Observer Object Interface & Classes

- **Observer**: An object interested in the event

```
public interface SomeObserver {
 void update(); }
```

- all observer objects must implement the same interface (even if they are diff types/versions that implement `update()` differently.)
  - so that the `subj0` can have **one list** of the interface type.
- a public `update()` method that, when called (by the `subj0`), executes some specific action in response.

- (only in most basic version) `obs0` encapsulates instance of the `subj0`

### Specific upgrades to the Observer DP

- \* observer registered to multiple subjects, wants to know which one called its `update()` method

**SOLUTION:** have the `update` method take a `subj0` as a parameter so that `obs0` receives this info when executing response action.

- \* observer wants **event context**: more info/details on the event that occurred, rather than just knowing that it took place

**SOLUTION:**

- 1) create an `Event` interface & various impl objects representing diff types of events & encapsulating specific info about those events
- 2) have the `obj0`'s `update()` method take an `Event` object as a parameter
- 2) have the `subj0`'s `notifyObservers()` method also take an `Event` as a parameter, so that it can pass this object into each `obs0`'s `update(Subject, Event e)` method when it calls them, choosing which `Event` to pass in based on what occurred.

## Unit 12: Functional Programming

### → What is Functional Programming?

→ functions as their own code "objects" — the "first-class citizen"  
being instantiated & utilized, passed around as arguments, modified,  
stored as variables, etc

- as opposed to OOP, where the focus is on creating objects (which are basically collections of data) & having them interact with each other.

→ the Observer DP is an example of a program that could be implemented with FP rather than OOP — the `obs` classes are effectively just functions since they are only used for their `update()` method.

### Functional Programming & Java

→ Java is an OO programming language (not made for FP projects)  
but they eventually added some language-level support (syntactic sugar) for FP — anon. classes & lambda expressions

## Unit 12: Anonymous Classes & Lambda Expressions

→ useful when we want to do functional programming but still have to operate within the realm of OOP & create object classes for the functions (like observer objects which represent `update()` functions)

→ purpose: for any interface, allows us to create a new class & instantiate an obj of that class — all in one expression!

- as opposed to creating a whole new .class file just to define a method (a "function") that's only going to be used once.

→ the expression creates a new class (that doesn't have its own name) & creates (& returns) only one single object of that class... we can't call this class later or create any ^ objects of it.

→ "expressions that result in a reference to an object that implements some interface."

### How do Anonymous classes work?

```
Main.java:
Fan tarheel = new Fan(); // creating a new instance of the interface type
@Override
public void update(Game g) { // defining the methods of the interface (aka what
 if(g.whosWinning().equals("UNC")) {
 System.out.println("Go Heels!");
 }
}
```

would be the "class body" right then & there

→ the anon classes do technically have names for purposes of identification, but these "names" are just random sequences of numbers chosen (and used only by) Java — so all of this is hidden from us b/c we don't have to care about that.

### How do Lambda Expressions work?

• they are basically just anonymous classes for interfaces that define only 1 single method.

```
Main.java:
Fan tarheel = (Game g) -> {
 if(g.whosWinning().equals("UNC")) {
 System.out.println("Go Heels!");
 }
}; // parameter list of the method.
```

→ only 1 method, so don't have to specify which one we are defining

## Leftovers (not in study guide)

- Test Driven Development
- .the fail() assertion method
- "Envisioning a .... Unit tests" & the algorithm specification stuff (all of those pages)
- design patterns
- the 3 strategies for making an iterator class
- entire Strategy DP chapter

## Decorator leftovers:

- unwrap() method
- limitations of the pattern

## Unit 11 leftovers

- benefits & criticisms of Singleton

## Unit 12

- alternate ways to support Observer programs w/ multiple events

# Graphical User Interfaces

What is Human Computer Interaction (HCI)?

- The original asynchronous programming model.
- part of the field of HCI.
- a major field of computer science (& a very active field of CS research) that focuses on the interesting problems & challenges of how we interact with computing devices
- a "computer" can be anything from a PC to a car, medical device, robot, or etc.
- they are essentially a **feedback loop** between a human & a computer;
  - \* A human provides some sort of interaction "input" to the computing device (like pressing a button)
  - \* The computer reacts to / "consumes" this interaction & then produces some feedback action to the human (like a sound playing, or a window popping up) ...which the human can then respond to with another interaction.
  - \* continuous cycle of the human & the computer providing & consuming "interactions" from each other.

What is a command line interface? (CLI)

- A text-based user interface where users interact with the computer by inputting lines of text (called "commands") into the "command line"
  - \* the shell a.k.a. the platform where a user can input commands
  - \* for ex, the Terminal application on Macbooks
  - pressing 'return', & having the computer perform some response action.

What is a graphical user interface? (GUI)

- A visual user interface where users interact with the computer through graphical components (such as buttons, menus, windows, and icons), instead of through typed commands (text-based)
- much more interactive than CLIs
- GUIs were first created in the late 1970s, & have essentially taken over the way that we think about computers.
- **GUIs are made up of UI components (widgets).**

What is a component?

- A UI element that acts as a unit of interaction
- basic UI components can be composed together to make compound **UI components** — like a menu, a pop-up/modal/dialog, or a panel (just some examples)
  - text label
  - image
  - geometric shape
  - Icon
  - button
  - Input/Slider
  - text input field
  - password field
  - hover tooltip

How do we use UI components to create GUIs?

- we rely on some 3<sup>rd</sup> party software (some library or operating system or etc.) to provide us the UI components, which we then pack into our program.

What was Java's original GUI library?

→ AWT (Abstract Window Toolkit)

→ it provided basic UI components in the form of classes (Button class, Slider class, etc.)

→ We write a program & call & utilize these AWT classes to abstractly design a GUI

• can't see any of the visuals while coding...only after running the code

→ Then, when we run the program, the Java runtime environment translates your code into an interface

displaying those user components, in a way that's specific to the operating system

- i.e., if you write a program on your Mac laptop for an interface window with buttons & a scroll bar, running the code generates (what looks like) an actual Mac application, with standard Macintosh format buttons, sliders, text etc.

- if you run the same code on a Windows computer, it would generate a Windows-style interface

- This aligns with Java's goal of being code that you could write once & run on any computer.

→ it could only provide the most common/basic UI components that it knew that every operating system would be able to support/have a built-in display mechanism for.

→ Limited to just the intersection of all of the different operating systems' UI toolkits

→ platform-dependent

→ Java Swing, a new set of classes that was an extension of AWT

→ Java designed its own GUI framework for how things looked & behaved

→ Intended for desktop first.

→ Rather than connect back/rely on some existing operating system component, the Swing components "draw themselves" & result in a Java-specific interface "look and feel"

- No matter what computer you run a Swing program on, the produced interface looks exactly the same (standard "Java Swing" format, rather than Macintosh, Windows, etc.)

→ PRO: not limited to only the basic components that already exist on operating systems.

→ Swing is still the built-in library provided by Java for writing GUI programs.

→ A modern 3rd party GUI framework / toolkit

→ Well-known (one of the "latest & greatest") & widely used

→ It allows you to write a GUI program that can arrange/reformat itself properly for different operating systems

- For ex, can write a program & run it on your laptop OR your phone — the components can render themselves to fit a smaller screen

→ the components in JavaFX are written to be responsive to their display environment & context.

→ JavaFX was influenced/inspired by web application development & was created with responsive design in mind.

How does AWT work?

What was the drawback of AWT?

What was the second version of Java's GUI framework?

How is Swing different from AWT?

What is JavaFX?

What is special about JavaFX?

What characterizes the visual appearance of JavaFX GUIs?

→ Like web dev., JavaFX enforces a separation of content -

which pieces you're putting together, & where they fit on the screen  
from style -

how those pieces look (in terms of color, format, etc.)

# Using JavaFX

What is the JavaFX "Application" class?

→ see "Graphical User Interfaces" notes for definition of JavaFX.

- an **abstract class** provided by JavaFX that serves as the "entry point" for creating a JavaFX program.
  - to create a JavaFX program, you must create a class which extends **Application**.

1. An overridden version of **Application's** abstract void "**start(Stage stage)**" method:
  - inside this method is where we actually set up the GUI - sort of the "main" for a JavaFX program

2. A **public static void main(String[] args)** method:

• this is where we run our program by calling **Application's** static **launch()** method

- An object that represents the window (of our GUI) on the screen that JavaFX creates for us.
  - **start()** takes a **Stage** obj. as a parameter because inside the method, we create our GUI by putting things on the stage.
  - the **Stage** is the overarching, most broad container for our interface but we don't directly add UI components to it... it is essentially just the space on your screen that was allocated for your application.

- Low-level/basic preferences, such as the title of the window (with **stageobj.setTitle("...")**), where it will pop-up, etc... **setResizable**, **setMaximized**, **setFullScreen**, **setMinWidth/Height**, **setMaxWidth**

- by creating a **Scene** object and adding it to the Stage using the **stageobj.setScene(Scene scene)** method.

- A **container** for the tree of components that we've created, that takes a **root UI component** (like a **Pane** object) as its input.
  - **Scene** has various constructors but, for example, **new Scene(Pane pane, int width,int height)** creates a new scene of the given size (in pixels).

- The idea behind JavaFX is to display UI components by putting them in "containers" (like **Pane** objects) & combining these containers together to eventually end up with a "tree" of UI components that then make up our scene.

• the "container" that contains everything (including other containers) is the **root UI component** that is then passed into a new **Scene** object.

- a UI component that acts as a 'container' & that we can use to describe where other UI components should be placed in our window
- since they are UI-components themselves, **Pane** objs can contain other **Pane** objs (thus creating the tree mentioned earlier.)

What are the (basic) components of an **Application** subclass?

What is "Stage"?

What features can we modify with the **Stage** object?

So how do we actually add things to the **Stage**?

What is a "**Scene**" object?

What is meant by "root UI component"?

What is a "**Pane**" object?

## How do Pane objects work?

Example of a **Pane** subclass?

→ You add UI components to a **Pane** as its "children";

`pane1.getChildren().add(button1)`

→ **Pane** has several subclasses that each position their children using different layouts

→ the **StackPane** object — stacks its children directly on top of each other

- basically you create a new **StackPane**, add UI components to its list of children, & then the **StackPane** object does the work of displaying the children in its subclass-specific layout.

→ We can have our program dynamically decide (based on some interaction within it) what it wants to display on the window, & it can swap out which pre-curated **Scene** object to put on the stage.

→ A good example of a basic UI component (to demonstrate how JavaFX UI component classes/objects work).

→ We can create a new button & then configure it using the various methods provided by this **Button** object class, such as `setText` (to add text) and `setBackground` (to set the color).

→ Just because a UI component was instantiated doesn't mean it will be displayed — to display it, we must put the component on our "**scene graph**" by adding it to a pane which eventually gets put onto the **Scene** & **Stage** of our GUI.

## What is a **Button** object?

How do we display the **Button** after we've created it?

Example of a GUI class using the components described so far?

```
public class myGUI extends Application {
 public void start(Stage stage) {
 stage.setTitle("Hello World!");
 StackPane pane = new StackPanel();
 Button avisBtm = new Button();
 btn.setText("avi");
 pane.getChildren().add(avisBtm);
 Scene scene1 = new Scene(pane, 300, 250);
 stage.setScene(scene1);
 stage.show();
 }
 public static void main(String[] args) {
 launch();
 }
}
```

1 Setting the stage title  
2 creating a new pane to hold the UI components  
3 creating a new Button component  
4 adding the button to the pane  
5 creating & setting the scene  
6 showing the stage on the screen  
7 the method that actually creates & displays our application

How do we attach actions to UI components?

→ currently, there is no action associated w/ the button — nothing would happen if the user pressed it.

→ JavaFX provides all the framework for visually designing a window, but when it comes time for our program to respond to a user interacting with a component, it needs us to "inject" that code — this is an example of **inversion of control**.

→ to implement this IoC, we utilize the **Observer design pattern**!

• the **button** (or other UI piece) is the **subject**, and our response code is the **observer**

In the context of the observer DP, what framework has JavaFX provided for coding response actions to UI interactions?

- Event Handler < T > : the interface for observer objects
- Button, Mouse, Slider, etc. : the subject object.
- setOnAction, setOnKeyPressed, setOnDragDetected, etc. : the subject object's registration method
- Event : an object that provides all the information on the action that occurred, including which subj0 caused it (recall event context)

\* various subclasses for specific event types, such as ActionEvent, MouseEvent, DragEvent, etc.

\* different UI components have diff event types associated with them

\* R5C4L1: In the observer DP that we looked at in Unit 12, we had the update() method take both a subj0 AND an "event" object as parameters, for purposes of providing event context... however, JavaFX chose to put all of that information into a single object (the Event obj.).

— therefore, handle() takes only an Event obj parameter... it can ask the Event object which subject UI component caused the event, if it needs to know.

→ a method provided by the subj0 class (the UI component) that basically registers observer objects (by taking them as a parameter) to its ActionEvents, specifically:

.setOnAction (EventHandler<ActionEvent> e)

\* similarly, setOnKeyPressed notifies all of its registered observers whenever a KeyPressedEvent occurs

→ the setOn\_ method basically says that "when \_\_\_\_\_ event occurs, here is the object that is going to handle/respond to it."

→ An interface defined by JavaFX for handling different types of events

→ This is the interface that all observer objects must implement!

→ it is a generic type (R5C4L1: parametric polymorphism) that each implementation observer fills in with the specific Event object that it is

public interface EventHandler < T > {

    public void handle ( T event );

}

"T" should be some type of Event object

the sole method defined by EventHandler < >

What is EventHandler < ?

→ where we actually code the response that we want to occur when the event occurs

→ equivalent to the update() method that we learned about in Observer DP.

→ takes the event obj as a parameter because it provides event context.

So how do we actually code an event response?

(example --> )

• since our observer "object" is only going to be used

once (and serves more as a function - R5C4L1: function)

Programming), we can use an anonymous class instead

of creating a whole new EventHandler < ActionEvent >

object class file!!

```
Button avisBtn = new Button();
avisBtm.setOnAction (
 new EventHandler<ActionEvent> () {
 public void handle (ActionEvent event) {
 System.out.println ("Hello!");
 }
 });
```

How could we make this syntax even tighter?

→ Since the `ObservedValue` interface only defines one method (`handle()`), we can use a lambda expression (`lambda`):

```
axisBtn.setOnAction((ActionEvent event) -> {
 System.out.println("Hello!");
});
```

## -Scene Graph-

What is the "scene graph"?

→ a tree of user interface components that is attached to a `Scene` object as the root.  
• typically, this root will be a `Pane` object.

→ There are several classes related to the scene graph: `Node`, `Parent`, `Region`, and `Pane`

→ the parent / "base" class of every component in the scene graph — aka all UI component classes

• `Button`, `Pane`, `Label`, `Slider`, etc. are all subclasses of `Node`.

→ Many of the JavaFX methods that we use to deal with our scene graph & its components take `Node` as a parameter, because the method is not specific to any one UI comp, but can work with any.

• We can just pass in the "type" of `Node` object (aka the UI comp) that we want to.

→ A subclass of `Node` that is the parent class of all of the UI-components which can contain children (subcomponents) — such as the `Pane` class.

→ represents an internal node of the scene graph.

→ a subclass of `Parent` & the parent class of all `Parent` UI-components that can be styled using JavaFX's implementation of Cascading Style Sheets (CSS).

→ a subclass of `Region` that is the parent class of all `Region` UI-comps that allow adding and removing children, and that specify how these children should be positioned on the screen.

## - JavaFX Layout Panes -

What are the 8 built-in layout subclasses of `Pane`?

|                           |                          |                         |                           |
|---------------------------|--------------------------|-------------------------|---------------------------|
| • <code>BorderPane</code> | • <code>VBox</code>      | • <code>GridPane</code> | • <code>TilePane</code>   |
| • <code>HBox</code>       | • <code>StackPane</code> | • <code>FlowPane</code> | • <code>AnchorPane</code> |

→ Each pane provides a different scheme for positioning its children on the screen.

→ A `Pane` that can contain exactly 5 children.

→ There are 5 regions where children can be positioned:

→ You can add one child to each area

→ Any area that doesn't get a child shrinks away & gets absorbed by the surrounding areas (when you display the window)

→ Any area that gets left over after children have been put in & the amount of space they can take up has been maximized, goes into Center — it will expand (horizontally or vertically) as necessary to take up any extra space.

| Top    |        |       |
|--------|--------|-------|
| Left   | Center | Right |
| Bottom |        |       |

What does **HBox** do?

- Children are positioned horizontally - either left-to-right or right-to-left - in the order that they are added to the Pane.
- Any extra unused space remains unfilled (aka white space in the window display)
- Same as **HBox** but vertically (either top-to-bottom or bottom-to-top)
- the default for **HBox & VBox** is left-to-right & top-to-bottom, but this behavior is configurable.

What does **StackPane** do?

What does **GridPane** do?

- Children are stacked on top of each other & centered in the window.
- Children are positioned in a grid with rows & columns, & the sizes of the rows & columns is calculated dynamically in order to accommodate the widest or tallest child in a row or column.
- Children are positioned left-to-right, top-to-bottom, starting in the upper left corner.
- Children wrap to the next row (or column) when the edge is reached.

What does **FlowPane** do?

- Same as **FlowPane** except every child's space ("tile") is forced to be the same size.
- To do this, it makes every tile to be the size of the biggest child.
- Children are "anchored" to the pane at a position respective to one of the edges of the window (top, bottom, left, right, or center)
- They can be anchored at absolute positions (like a co-ordinate point) or relative positions (like "25% away from the left edge" or "20 pixels down from the top edge").
- The most flexible of all the layout panes.

### - JavaFX UI Components -

What are some basic useful UI components in JavaFX?

- |                                              |                                          |
|----------------------------------------------|------------------------------------------|
| → <b>Label</b> - for displaying text         | → <b>CheckBox</b> - a checkbox           |
| → <b>Button</b> - a clickable button         | → <b>Rectangle</b> - a colored rectangle |
| → <b>ToggleButton</b> - toggleable button    | → <b>Circle</b> - a colored circle       |
| → <b>TextField</b> - a text input box        | → <b>Slider</b> - a slider bar           |
| → <b>Imageview</b> - for displaying an image |                                          |

## Style in JavaFX

What is the common programming pattern/approach for GUIs?

What is "content" code?

What is "style" code?

How do we separate style code in Java / with JavaFX?

What is a stylesheet?

What language do the style sheets use?

What is a style class?

- To enforce a separation of style code from content code.
- This principle emerged from & was influenced by Web programming/development principles.
- Defining what gets displayed : what UI components we will have, where we will place them, what layout we will use, etc.
- This is the code that goes in our `App` (extending `Application`) class file; setting the scene etc.
- Defining how to display the program contents — colors, fonts, padding, margins, etc... how we want everything to actually look
- If I create & run a program w/o any style, it will just look super ugly & probably won't make any sense to the user
- We are going to put the code that "declares" our style in a completely separate file (or set of files) called **style sheets**.
- In a Maven project using JavaFX, the...
  - \* Content code goes in `/src/main/java/...` (the specific app class inside the `java` folder)
  - \* Style code goes in `/src/main/resources/style/...`
- A file where we declare certain style rules / configurations, grouped into **style classes**, that our UI components can then adopt / "subscribe to"
- We use these in order to get our UI components to look how we want them to.
- A set of style rules can be associated with other structures as well, but for the purposes of this class, we will enforce "styling by class" & will only need to group rules into classes.
- **Cascading Style Sheets (CSS)**, which is a design language used to style UI components in style sheets.
- not coded in Java
- The style sheet files are `.css` files (rather than `.java` like usual classes)
- a designated block of code (that is given its own name) inside a style sheet that defines a set of style configurations.

## Example of a style sheet?

### main.css

```
.layout {
 -fx-background-color: #faf8e9;
 -fx-font-family: Arial, sans-serif;
 -fx-font-weight: bold;
 -fx-font-size: 12px;
 -fx-padding: 10px;
}

.scoreboard {
 -fx-alignment: center-right;
 -fx-spacing: 10px;
}
```

- the name of the style class, "layout", indicated by a `.`

- JavaFX defines different identifiers, such as `background-color`, `font-weight`, etc. as settings that can be set for its UI components
- put a colon (`:`) after the identifier name.
- All identifier statements begin with "`-fx-`".
- One .css file can define multiple "style classes" for multiple components.

How do we get our UI-components to incorporate a style class?

→ every Node (aka every UI-comp) contains a list called by the method `.getStyleClass()`, b/c it can be associated with multiple style classes (not limited to one) that can all be applied at once

→ Attach a style class to a Node using `Node.getStyleClass().add("name of style class")`

→ We have to associate the overall scene with the particular style sheet that we are using, so instead of `getStyleClass`, we use:

```
Scene.getStylesheets().add("styleSheet1/main.css");
```

↳ we input the file path from the resource subfolder

What happens when you add a style class to a `Pane`?

→ Although `Pane` objects are just containers that won't be affected by most style rules (like font & etc.), we can associate them with a style class if we want all of its declared children to adopt that style into their list of style classes.

→ If a UI-component has its own specific style rules that clash with those dictated by its parent `Pane`, the UI-comp's own rules take precedence & overwrite the pane's.

→ If the competing settings are on the same "level" (coming from the same `Pane`, `Scene`, or other UI-comp's `getStyleClass` list): JavaFX has a well-defined set of rules that decides which style settings get prioritized.

What happens if a UI-component has contradicting/competing style rules?

## Model-View-Controller

= crucial to the definition of MVC

What is Model View Controller?

- A software design pattern used for structuring and organizing programs for applications with a user interface (GUIs).
- MVC is an application-level pattern; it provides the architecture for an application as a whole.
- MVC started in the 70s-80s when GUIs were first developed, & has remained popular ever since.
- MVC can be used in desktop, mobile, and web applications.
- **Observer!** MVC is effectively a series of observer relationships between different parts of an application that are each responsible for specific aspects of an event-based GUI program.

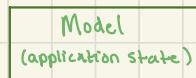
What other DP does Model View Controller employ?

What is the big idea behind MVC?

- To view an application as having 3 parts:
  1. The state of the application — the actual information that our app is trying to store & manipulate
  2. The way that our application looks/is presented to the user, & how the user interacts with it.
  3. A way to interpret & translate the user interactions (component #2) into manipulations of the underlying application state (component #1) — This is the idea that MVC introduces; to separate an app's UI code from its state management code

How does MVC execute this idea?

- By having each of these 3 components have their own separate, well-defined interfaces and responsibilities.
- The Model classes and View classes are decoupled from one another (no inheritance or direct references to class names), and the Controller provides the level of interaction between the two so that they can remain independent of one another.
- the exact way that the components are related to each other depends on the pattern being used — the classic MVC approach or the "alternate" approach.



\* Knows how the application works, but not how to show it to the user



\* Knows how to show the application to the user, but not how the app works.



\* translates View events into Model commands.

What is the advantage of decoupling the **View** from the **Model**?

- We can then replace the "View" component of our app seamlessly, without having to change anything about the Model.
- This is useful because we can make different **Views** that are compatible for/ specific to different UIs - like an iPad versus a phone, a computer, an audio-only interface, etc etc
- To understand MVC, let's consider trying to program a GUI for the game 2048 (the swipy game)
  - notes referring to the example are **in purple**.

What is an example application of MVC?

### - The Model -

What is the "Model"?

- the classes/partition of the program that stores the application state (**the current status of the application's operations**) & knows how the app **works**
- The **Model class(es)** are usually subject objects that get observed, because the rest of the application needs to know when states have changed.
- provides algorithms for data manipulation;
  - the numbers that should be displayed on the board
  - the current & best score of the player
  - whether a given tile is empty, & if not then what number it contains
  - The actual alg for combining & adding new tiles
  - And more!
- Yes! The **Model** classes should actually be able to work without a user interface at all - like, if you were to create its objects & run its methods/algorithms in the **Main** file, it should work as intended
  - for ex, **RECALL ADVENTURE** - we were able to "play" the game we created by running **our program in Main**, although there was no visual representation of it (but we could add one)
- the Model object doesn't assume or know anything about how it might be used (like on a UI level)
  - 1) **To encapsulate the application state** (in private fields)
    - int[ ][ ] board
    - int highScore
    - int score
  - 2) **To expose methods for accessing the states** (Controller will use these.)
    - getTile(int x, int y)
    - isGameOver()
    - getCurrentScore()
    - getBestScore()

Is the **Model** independent from the rest of the program?

What are the 4 main responsibilities of the **Model** object?

What are the 4 main responsibilities of the Model object?  
(continued)

3) To expose methods for modifying the states (Controller will use these)

- swipeLeft()
- swipeUp()
- reset()
- swipeRight()
- swipeDown()

→ the implementation of these methods is where the Model "modifies itself" & updates its state — for ex, swipeUp():

Comparing if the 2 tiles match, if so "combining" them by changing the values of the respective cells in board, as well as updating score, etc.

4) To notify its observers when a state has changed (like the game being over, a player's turn ending, change in the score, etc.)

- do this with a `notifyObservers()` method.

### - The View -

What is "View"?

- The part of the program that knows how the application looks
- creates & displays the user interface
- The package with View classes must be the place where we have all of our JavaFX code (since JavaFX is our UI library)

1) To create & generate the user interface, using the current state/data values encapsulated in the Model.

2) To refresh the UI whenever the application's state changes.

→ by acting as an observer object, observing either the Model or the Controller (depending on which approach we are using) for state changes and then updating how it looks accordingly

- for example, if the score changes, the View should be notified of this change so that it can update the number being displayed on the screen as the "user score".

3) To observe for user interactions & report them to the Controller (by calling its methods).

→ The View must report all user interactions to the Controller so that it can interpret those interactions in terms of what they mean for the application, and then execute the appropriate actions to update the application state (aka the Model), as well as update the View itself, if needed.

→ Define an interface for all of our View classes that contains a `render()` method:

```
public interface FxComponent {
```

    Parent render();

3

- the `render()` method generates and returns a scene graph representing the UI tree for the view.

→ `render()` usually returns a `Pane` object (RECALL that `Pane` is a subclass of `Parent`)

→ inside of the `render()` implementation is where we will actually create the UI JavaFX components and add them to our `Pane` container.

What are the 3 responsibilities of the View object?

How does the View refresh the UI?

Why must the View report user interactions to the Controller?

What is the suggested pattern for View classes?

What is the purpose of this suggested pattern?

- We can build up our final GUI (the one that gets displayed to the user via the **Stage** and **Scene**) **out of multiple View classes**;
- We can create "higher-level" **View component classes** that encapsulate several other **View classes** — by asking them to "render themselves" (with the `render()` method) — & then sort of render them all into a collection (a "View" that contains all of those smaller "views", arranged in a specific way)
- Essentially breaking our user interface into a logical hierarchy, & then creating a **View component class** for each level/part of that hierarchy — each of which knowing how to render itself — and combining them to build up our GUI
- in addition to rendering themselves, **all View component classes** (even lower-level ones that get encapsulated into higher level ones) need to **encapsulate a reference to the Controller object** in order to fulfill responsibility #3
  - this is how the connection between **View** & **Controller** gets made!
  - done in the **View class' constructor**.

What is the 2<sup>nd</sup> thing that **View classes must do**?

Example of a basic View component class?

```
public class ButtonView implements FXComponent {
 private Controller controller;
 public ButtonView(Controller controller) {
 this.controller = controller;
 }
 public Parent render() {
 VBox layout = new VBox();

 Button button = new Button("click me!");
 button.setOnAction ((ActionEvent event) -> {
 controller.handleClick();
 });
 layout.getChildren().add(button);
 return layout;
 }
}
```

the View interface (see prev. page)

encapsulating reference to its associated Controller object via the constructor

this View component knows how to render itself by creating a **VBox** (recall some subclasses) layout, putting a button in it, & returning the layout

the UI event of the button being clicked is forwarded to the controller by calling controller's `handleClick()` method to respond to the event.

`render()` returns the generated scene graph.

→ This component is simply just a **VBox** with a **button**, but it can be put inside of another component that contains it and other similar small components to create a **compounded component**.

• the compounded component can be put into another one, and so on... this is how we build up our final GUI — a tree of many **View components**.

Example of a compounded View component class?

```
public class CompoundView implements FXComponent {
 private Controller controller;
 private FXComponent leftPanel;
 private FXComponent rightPanel;

 public CompoundView(Controller controller) {
 this.controller = controller;
 this.leftPanel = new LeftPanel(controller);
 this.rightPanel = new RightPanel(controller);
 }

 public Parent render() {
 HBox layout = new HBox();
 layout.getChildren().add(leftPanel.render());
 layout.getChildren().add(rightPanel.render());
 return layout;
 }
}
```

This component is a combo of 2 other View components... creates a new LeftPanel & RightPanel Component & ties them to the same Controller object that it is tied to.

CompoundView renders itself by putting those 2 encapsulated View components into a layout, and asking them to render themselves

- Since this component doesn't create any new UI components & simply combines other View classes, it doesn't need to directly forward any interactions to the Controller — that has already been taken care of by the internal components.

### -The Controller -

What is the "Controller"?

- the part of our program that handles user interactions by calling appropriate methods from the Model classes.
  - The Controller is the "brains" of the operation — it contains all of the higher-level logic about how to actually use the Model's methods in a meaningful way.
  - it bridges the gap between the low-level interactions with the smaller low-level View components, & what they mean at that moment and given the current state of the app.
  - The Controller consists of methods that translate user interaction events into commands for the Model.
1. A user interaction occurs on the GUI and the View component reports this occurrence to the Controller by calling one of its methods.
    - user swiped an [8] tile onto another [8] tile
  2. Controller interprets that interaction into what it means in the context of the app & the application state — aka translates it into manipulations of the underlying Model.
    - the user is trying to combine the tiles, & wants a new combined [16] tile to appear

What is the process/event sequence through which the Controller gets used to report updates to the Model?

3. Controller interacts with the Model to execute these interpreted actions by calling its methods so that it can "update itself."
- controller calls Model's swipeUp() method — which may take a board co-ord as a parameter (like the co-ord of the tile that was swiped to — this is just a rough abstract example)
  - The actual job of determining whether the 2 tiles are the same, combining them, updating the board to have a [16] where there used to be an [8], increasing the user score etc... is all done in the Model's method implementation — Controller simply has to set off the spark to let Model know what has happened.

What does the Controller class need to have?

Example of a Controller class?

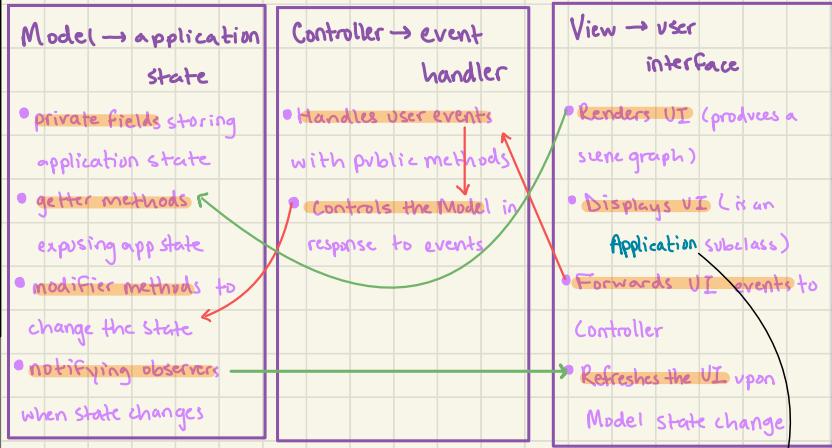
→ must encapsulate a reference to the Model so that it can "manipulate" it & call its methods!

```
public class Controller {
 private Model model;
 public Controller (Model model) {
 this.model = model;
 }
 public void handleSwipe (Direction dir) {
 switch (dir) {
 case UP:
 model.swipeUp();
 case DOWN:
 model.swipeDown();
 case LEFT:
 model.swipeLeft();
 case RIGHT:
 model.swipeRight();
 break;
 }
 }
}
```

What is the event sequence through which the Controller gets used to report updates to the View?

→ This sequence differs based on which approach of MVC we are using — classic or alternate. We will discuss this in a sec.

**Recap:** What is the role of the Model, View, & Controller?



RECALL: JavaFX

### - Classic MVC versus Alternate MVC -

What do classic & alternate MVC have in common?

→ The strategy for sending information from the View to the Model is the same for all MVC approaches — see the red-highlighted notes under "Controller" on page 128

- indicated by the red arrows

→ But the strategy for reporting Model updates to the View is where these 2 approaches deviate.

→ indicated by the green arrows

→ View components register themselves as Observers of the Model so that they can be notified of state changes (which, as we know, are inflicted upon Model by Controller), and so that they can then update themselves (aka "refresh the UI") to reflect these changes.

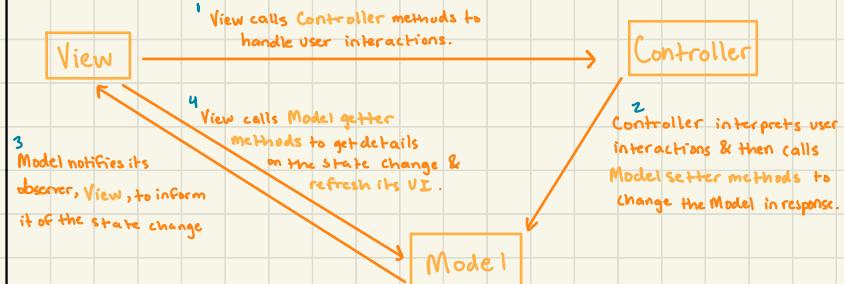
→ Since View is an observer of Model, it gets notified anytime a state change occurs. Knowing this, it can then get the specifics on which states have changed by calling the Model's getter methods (like in the View's update() method (RECALL: Observer DP))

• Alternatively, we can design the Model's notify() and the View's update() methods to pass in parameters that specify the state change (RECALL: Event (context)... this type of small design choice is up to us in terms of how to implement it.

What is the classic MVC approach?

How do the View components know which changes to reflect?

Overview of the flow of information in a classic MVC pattern?



What does the setup for a classic MVC look like?

```
Model model = new Model();
Controller controller = new Controller(model);
View view = new View(controller, model);
model.addObserver(view);
```

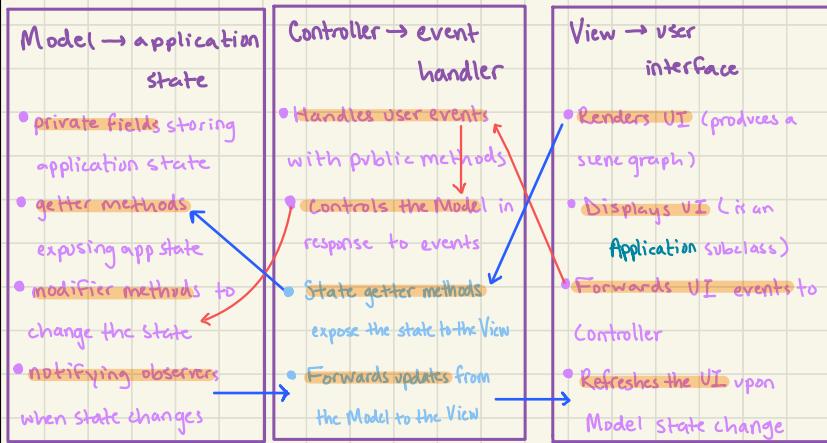
Controller contains a reference to the model so that it can interact with the Model in order to update it.

add view as an observer of the model obj  
it encapsulates a reference to

View components know about both the controller AND the model so it can forward interaction events & receive app state change notifications, respectively.

What is the alternate MVC approach?

- The Model and the View are fully decoupled — they don't even know that the other exists.
- The Controller sits in between them & provides View with methods to retrieve info from the Model.



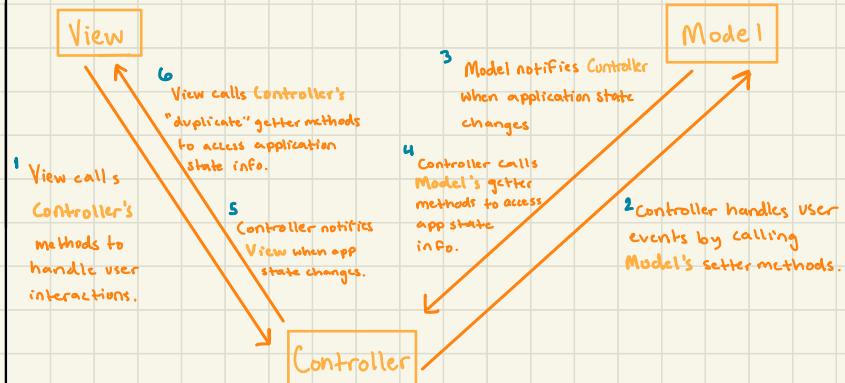
What are the observer relationships in the alternate MVC?

- The View component only has access to the Controller component, & registers as an observer of Controller.
- Controller registers as an observer of Model (instead of View) & is notified anytime an application state change occurs.

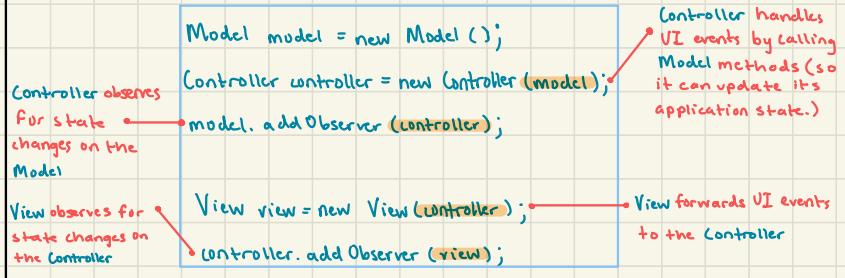
1. State change occurs in Model
2. Model notifies Controller of the occurrence
3. Controller calls Model's "state exposing" (getter) methods to get event context on the change that occurred
4. Controller, as a subject object, then turns around and notifies View of the state change occurrence it was just informed of! ("forwarding the updates")
5. Controller contains its own "state exposing" (getter) methods that essentially "duplicate" those of the Model (it gets the data to fill these methods using Model's getter methods.)
6. View calls the Controller's getter methods for event context, & then refreshes its UI with respect to the app state changes.

How does View get informed of state changes?

Overview of the flow of information in an alternate MVC pattern?



What does the setup for the alternate MVC look like?



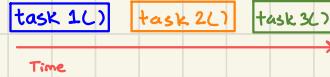
→ We could even go so far as to not have the View observe the Controller at all, & instead have the Controller respond to event updates from the Model by turning around and directly manipulating the View itself! aka Controller completely in charge.

What is the other way to execute the alternate MVC approach?

# Concurrent Programming

What is the **sequential computing** model?

→ A way of modeling your computation such that, when a series of computations are executed one at a time, each computation must finish before the next can begin.



What is the **concurrent computing** model?

→ When a series of computations are executed during overlapping time periods.



→ There are two ways that a program can exhibit concurrent computing to have multiple actions be executed at the same time:

- **in parallel**: actually being able to execute multiple tasks simultaneously; aka **parallel programming**.
- **by context switching**: The computer is only actually performing one action/task at a time, but it switches back & forth between them (rapidly), so that they appear to both be making progress at the same time;

Kind of a "false appearance" of concurrency because there is no actual increase in performance or decrease in time spent.

What is the **Parallel Computing Model**?

→ When multiple tasks can simultaneously be executed on separate processing elements

→ "parallel computing/programming" is sort of a branch / type of concurrent computing

2. Parallel programming
2. asynchronous programming

What are the two forms of concurrent computing?

What is a "model of programming"?

→ a way of programming a "model of computation"

→ You can mix & match these, but ultimately you are limited by your resource - for ex, even if you implement an asynchronous programming model on a sequential computer, the program will never actually go any faster, because it is still only doing one thing at a time.

What is synchronous programming?

- A model of programming where a task may be started, and the program has to wait for it to complete before continuing to run.
  - this is the "normal" way that we have thought about/considered our programs thus far - for ex, calling a method: it stops execution of the current file to go execute the code in the other file/method



What is asynchronous programming?

- A model of programming where a task may be started, but the program continues on without waiting for it to complete
  - the program can later coordinate with that task (to see if its complete, get its result, etc.)

### - A History of Computer Processing Speed -

What is "Moore's Law"?

- This dude in the 70s realized that the number of transistors that could be fit on a processor — which, in that time period, was directly related to how fast it could do computation — was increasing at a very specific rate — it was doubling every 18 months.
- Moore's law: Computing speed (a.k.a. the speed of the fastest/most recent computer developed by humanity thus far) doubles every 18 months.

→ Shockingly, this law has held true for the last 40 years (and counting)!

- in the 1970s: lithographic techniques for etching silicon improved (so transistors got smaller), but this technique was eventually exhausted (how much smaller could they possibly be, yk...)
- in the 1980s: we were able to make processors faster (in terms of time) but this made computers really hot because they generated a lot of heat (rocket Macrol) so this technique was also eventually exhausted.

→ in the 2000s: the multicore revolution!

- instead of designing new processors that were faster than all of the ones before it, we started architecting a larger number of smaller processors that are like mini "parallel" computers, each of which had smaller cores that were then tightly integrated into a shared memory and cache.
- aka, we made our computers faster by allowing them to do more than one thing at a time

What is the multicore revolution?

- the multicore revolution is what is driving Moore's law today; the reason why we continuously have increases in computer performance is because we are developing increasing parallelism in our designs.
- can be up to 2000 cores in one processor (prob even more)
- it is the key to why our (society's) computers get any faster!
- Multicore can only increase computer performance if we can find ways to parallelize our tasks.

### - Parallel Programming -

In what situations can multicore **easily** be used to increase performance?

Example?

How was this done before multicore?

When is it more challenging to integrate multicore for increasing performance?

- for a computer's **completely independent processes**, where there are no data or logic dependencies, or aggregating data operations.
- A laptop and its applications : Chrome & Spotify & Word & the operating system can all run at the same time because the laptop can have a separate core running each program.
- You could still run multiple programs at the same time, but they weren't actually being run in parallel - the operating system was just rapidly switching between the multiple tasks, giving a "false appearance" of concurrency... see context switching notes 2 pages ago
- no increase in performance because no time is saved. The overall time it takes is still going to be the sum of the time it takes to run each program individually.
- When we are interested in improving the performance of one particular program (not as a part of a collection of programs)
- With one program, it is more complicated to find ways to "parallelize the tasks"; being able to get one part of the program started while another continues to run and getting them to coordinate/exchange answers etc. when it's done, for example.

### - threads and multithreading -

What is multithreading?

What is a "thread"?

What information is encapsulated in a thread?

- this idea of writing a program that is operating in more than one place at a time
- an abstraction for executing a program -- to execute any program, your operating system creates a thread.
- 3 things:
1. **Instruction Pointer**: the current point of execution.
    - tells us where we are in the program.

2. **Call Stack:** Which methods are currently executing

- holds the structure of "call frames" that build up as we progress through the program.
- tells us where we have to go back to when we return from a method (RECALL: exception handling; see notes on call stack, pg. 58)

3. **Memory:** the contents of the memory (objects and etc.), including the heap.

→ basically, a "thread of computation" is comprised of where you are in the program, how you got there, and the current state of the memory.

→ Each thread has a separate instruction pointer, separate call stack, and a shared memory (they all share the same heap)

How are these components represented in a multithreaded program?

How do we communicate between threads?

→ by coordinating the use of shared memory, since that is the one thing they have in common.

## Java Support for Concurrent + Parallel Programming

What is the "Runnable" interface (and object)?

- the key mechanism to writing multithreaded programs in Java.
- the **Runnable interface** is built into Java and defines objects that represent a task that can be performed.
- Runnable objects are basically "mini-programs" – an execution of a certain task that can be run by a thread.
- the **Main class** in our Java programs is actually a **Runnable object**!

```
public interface Runnable {
 void run();
}
```

How do we execute the task defined by a **Runnable** object?

Example of a "Runnable" object?

- by calling the object's **run()** method – the only method defined by the interface.
- **run()** contains all the code that we want executed. It doesn't take any parameters or return any results.
- (RECALL: lambda expressions) since **Runnable()** only defines one method, it can be instantiated using a **lambda expression**!

(inside of some other class)

```
public static void main(String[] args){
 Runnable task1 =
 () -> {
 for (int i=0; i<10; i++) {
 System.out.println(i+1);
 }
 };
 ...
}
```

→ created using a lambda expression

Fig. 1

- Just creating a **Runnable object** doesn't do anything – if we ran the class that this code is inside of right now, nothing would happen (or print). We've simply created the object & assigned it to the variable "**task1**".
- Both! **Runnables** can be used either synchronously OR asynchronously, hypothetically.

Which model of programming are **Runnable** objects used in?

How do we run a **Runnable** object synchronously?

Example?

- by simply calling the method (**run()**) on that object, like you would with any other object.
- add the following lines of code to **Figure 1**:

```
System.out.println("We are printing 1 to 10");
task1.run();
System.out.println("Done!");
```

→ The console will print "we are...10"; followed by the print statements made inside of **task1**, and finally, "Done".

- There is no concurrency here: we simply created a **Runnable object** and executed its **run()** method.
- The program is still only doing one thing at a time.

How do we use a **Runnable** object **asynchronously**?

What is the **Thread** object?

How do we create and run a new **Thread** object?

→ Using Java's built-in **Thread** class!

→ it represents a thread of execution; it allows you to create new threads that can then be run at the same time! Hence, asynchronous programming.

→ When wanting to multithread, we always start in/with one "main" thread that always starts in our main method ... this is where our main program starts and where we start new threads, if we want to.

→ When we create a new **Thread** object, it needs to be told what to do. Thus, the constructor takes a **Runnable** object as a parameter:

- `Thread thread1 = new Thread(task1);`

→ NOTE: multiple **Thread** objects CAN use the same **Runnable** object - it would just be 2 independent executions of the same task.

→ To start/run the thread (& thus begin execution of the code inside the **Runnable** obj), call the thread's "start()" method

- `thread1.start();`

→ Once we have called `start()`, our program is effectively running in 2 places at the same time (aka concurrently and asynchronously)!

- the main thread continues to execute. Meanwhile, your operating system and the JVM conspire to create a second thread of control with its own stack

\*the second thread begins inside of the **Runnable** obj - that is its version of "main"

```
public static void main(String[] args){
```

```
 Runnable task1 =
```

```
 () -> {
```

```
 for (int i=0; i<10; i++) {
```

```
 System.out.println(i+1);
```

```
 };
```

```
 System.out.println("We are printing 1 to 10");
```

```
 Thread thread1 = new Thread(task1);
```

```
 thread1.start();
```

```
 System.out.println("Done!");
```

Fig.2

At this line, the new thread branches off and starts executing the `run()` method of its given **Runnable** object - at the same time as the rest of the program.

→ basically, we can't make any assumptions about how fast the offshoot threads are or which one will finish first.

\*if there are multiple threads, we also can't make assumptions about the order in which each of the threads' tasks will occur in respect to one another

→ Usually (but this is NOT a rule), the print statement in the main method ("Done!") will end up getting executed first... but we can't just assume this.

So what order will the statements print in?

Example to demonstrate  
the order of execution with  
multiple threads?

Fig.3  
public static void main(String[] args){

Runnable task0

↳ → {

→ for (int i=0; i<10; i++) {  
    System.out.println(i);  
    System.out.println(" ");  
}

Thread thread1 = new Thread(task0);

thread1.start();

Thread thread2 = new Thread(task0);

thread2.start();

System.out.println("Done!");

1. our main thread created thread1 and started it; new execution has begun at the "→"

2. new execution has begun at "→" ... running same code as thread1, but completely independently

3. The main thread continues on its own way as well, right after starting the other threads.

What order will these statements print?

→ Not guaranteed to be the same every time, but one possible output:

Done!

00 1 1 2 3 2 4 3 5 4 5 6 7 8 6 7 7 8 9

- The main thread immediately goes off to print its statement before the other threads even have a chance to.

- However, the timing and order of each offshoot thread isn't consistent in any way...it's just sort of random.

→ we can accomplish this using the Thread object's join() method!

→ Waiting for a previously spawned task to finish is a common thread coordinating operation.

• Ex: think of sending 1 thread to "mix dry ingredients" and one to "mix wet ingredients"...

main method cannot "mix wet & dry ingredients" until those 2 threads have finished

→ When it reaches a line of code that calls join() on one of its (obviously already declared) offshoot threads, the main method pauses/stops execution until the thread is done executing the code in its Runnable object.

→ adding the following lines to Figure 3:

thread1.join();

thread2.join();

Main() pauses until thread 1 is complete...

thread2 still runs & is completely unaffected by this line.

After this line has given Main() the go-ahead to continue, it now "pauses" until thread2 is complete. thread2() might have already been long-finished by this point (in which case the .join() call will signal to Main() that it can keep running), but .join() basically functions as a stopper/checkpoint to make sure that Main doesn't continue until the thread's task is done.

Example?

But what if we want the Main thread to wait for the other threads to finish?

How does the join() method work?

# Java Support: Mechanisms for Thread Coordination

RECALL: what is "join()?"

→ Process of waiting for a previously spawned thread to finish its job before continuing your main program.

→ `join()` is the simplest/most basic form of thread coordination.

→ A segment of concurrent code where the timing of the execution (of the 2 or more pieces that are running concurrently) affects the result.

→ When 2 or more threads are actively sharing memory - aka reading from or writing to the same object.

- "writing to": manipulating, using, or otherwise modifying the same obj
- "reading from": retrieving state info from the object

→ When this happens, we have a race condition because we have to make sure that 2 threads aren't manipulating the same object at the same time.

→ Which thread's mutation of the object will actually be in effect? We don't know, because it depends on which thread executes first.

• could end up with undesirable results.

→ If one thread reads a field but then another thread overwrites it, the value read by the one thread is wrong (aka state valves) because it doesn't see the modification made by the other thread.

→ imagine a Counter class that increments & decrements an integer:

public class Counter { Fig.1

private int num;

public Counter() {

num = 0;

3

public void addOne() {

num = getValue() + 1; 3

public void subtractOne() {

num = getValue() - 1; 3

public int getValue() {

return num;

33

encapsulates an integer, which it starts at 0

These lines of code actually each perform 3 operations!  
(this is important):

1. gets the current value of num
2. adds/subtracts 1 from it
3. effectively sets the value by assigning the new decremented/incremented number as the value of the private field num

when addOne() or subtractOne() is called, the state of the class changes

OK, so how will we use threads

with this example class?

→ In the main method, we will have 2 threads that both use the same Counter object:

Counter counter = new Counter(); Fig.2

Thread thread1 = new Thread(() → {

for (int i = 0; i < 1000; i++) {

counter.addOne(); 3

});

One thread increments the counter 100,000 times

(continued on next page)

```
Thread thread2 = new Thread(() -> {
 for (int i = 0; i < 1000; i++) {
 counter.subtractOne();
 }
});
```

the other thread decrements the counter 100,000 times

```
thread1.start();
thread2.start();
thread1.join();
thread2.join();
System.out.println(counter.getValue());
}
```

starting both threads simultaneously

waiting for both threads to be done, and then printing the value of num

What will be printed by the system?

→ Logically, we should want the system to print 0, since we added & subtracted 100,000 from the value.

→ However, we actually end up with a different number every time we run the program!

-30,654      -2,782      5293      ...

→ Imagine that `num=0` and `addOne()` and `subtractOne()` are called concurrently:

#### Thread 1

1. Gets the `num`, which is 0.
2. adds 1 to that number
3. sets `num` equal to 1

#### Thread 2

2. Gets the `num`, which it also finds to be 0, because it began at the same time as thread 1 (not after!).
2. Subtracts 1
3. sets `num` equal to  $0 - 1 = -1$

Which one of these is going to happen is a race condition! depending on which thread "wins" the race — aka executes first — `num` will either be -1 or 1 (if we were to call `.getValue()` at this point)

→ basically, only one of the threads' actions takes effect at a time... and there's no way to know which. It's just up to chance.

→ because they will both start with the same value, try to modify it in different ways, & then we have an unspecified race condition for what will actually end up happening.

→ By marking specific methods (of a class) as synchronous, aka making them "mutually exclusive"... using the "synchronized" keyword.

• By marking an object's methods as synchronized, we are saying that none of those methods can be executed at the same time (like by multiple threads).

Conclusion: why don't we want 2 threads writing to one object simultaneously?

How do we avoid these unspecified race conditions?

What Java feature do we use to enforce mutual exclusion?

→ The "synchronized" keyword, which is Java's syntactic sugar (recall: for-each loops)  
for every object's mutual exclusivity locks (more on this ahead)

→ the keyword is placed in front of a class' method definitions:

```
public class Counter {
 private int num ;
 public Counter() {
 num = 0; }

 public synchronized void addOne() {
 num = num + 1; }

 public synchronized void subtractOne() {
 num = num - 1; }

 public synchronized int getValue() {
 return num; }
}
```

→ Java ensures that no two synchronized methods of a given instance of the class will ever be executed at the same time by different threads.

What does it mean that synchronization is "object-specific"?

→ Basically, this mutual exclusion isn't applied to the class as a whole, but to each specific instance of the class;

• if we have 2 instantiated Counter objects, for example, then it is okay to have a thread inside of addOne() for one of the objects and simultaneously another thread in subtractOne() of the other object... but 2 threads could not be occupying both of those methods in a shared object at the same time.

What does thread execution look like with synchronized methods?

→ When 2 threads want to be in a synchronized method at the same time:

1. Whichever thread started its action / got to the method first will be allowed to continue into the method.
2. The thread that gets there second will be suspended, told to sit and wait.
3. The second thread will enter the method as soon as the first thread has left it.

→ refer to the following part of Figure 2:

- thread1 calls addOne() (in its Runnable object) and thread2 calls subtractOne()  
... but since these 2 methods have been marked as mutually exclusive ("synchronized"), they will not execute simultaneously.

```
thread1.start();
thread2.start();
thread1.join();
thread2.join();
System.out.println(counter.getValue());
```

What will be printed by the System?

→ Since none of the addOne() or subtractOne() calls happened at the same time, there is never a race condition! every addOne() call is matched up with a subtractOne() call, so the system always prints 0.

Does "synchronizing" methods affect performance?

→ Since setting up these locks of mutual exclusivity does force synchronous/sequential programming (as opposed to concurrent /asynch) for parts of your program, it does result in some overhead in terms of speed /performance.

- But, this is a necessary qualification

What type of operation is the "synchronized" keyword?

→ Like mentioned before, "synchronized" is Java's syntactic sugar for enforcing mutual exclusion of methods marked by the word.

- Behind the scenes of this term, the JVM is doing some more intricate task to achieve synchronization.

What are some best practices involving synchronization?

→ every field in a class that reads or writes field values should (usually) be synchronized.

→ Try to keep synchronized methods as short as possible

- this is because they restrict threads to working one at a time, rather than concurrently (so performance speed gets reduced).

So how does Java achieve synchronization?

What is a Lock?

According to Oracle's documentation

→ By using something called a Lock

→ A tool for controlling access to a shared resource by multiple threads

→ Commonly, a lock provides exclusive access to a shared resource; only one thread at a time can acquire the lock, and all access to the shared resource requires that the lock be acquired first.

→ Every lock has the ability to either be "locked" or "unlocked", and to be set to be one or the other of these two states.

→ basically, when a method is marked as synchronized, the JVM internally creates a single ReentrantLock for every instance of the synchronized class (like Counter, for ex)

→ The one lock is shared across all synchronized methods of the class.

→ The synchronized keyword basically says, to any thread that tries to enter its method:

- "before this method is allowed to run, you have to obtain/acquire the lock corresponding to this object instance. If you can't acquire it, you have to wait."

- "if you are able to acquire the lock, you can proceed into the method and then release/unlock the lock as you are exiting the method."

→ Whichever thread gets to the method first "gets the lock" and is allowed to proceed into the method. The other methods then must wait for the lock to be available.

How does Java use the locks?

What does the "behind the scenes" implementation of synchronized actually look like?

```
public class Counter {
 private int value;
 private Lock lock;

 public Counter() {
 value = 0;
 lock = new ReentrantLock();
 }

 public void addOne() {
 lock.lock();
 value = getValue() + 1;
 lock.unlock();
 }

 public void subtractOne() {
 lock.lock();
 value = getValue() - 1;
 lock.unlock();
 }

 public int getValue() {
 lock.lock();
 int v = value;
 lock.unlock();
 return v;
 }
}
```

- every Counter instance has its own Lock
- "ReentrantLock" is the specific lock implementation used for the synchronized keyword operation.

#### Steps for threads modifying the object:

1. Acquire the lock, waiting until it's available if necessary  
(lock.lock() will only run if the lock is available)
2. (critical section (executing all method-related code))
3. Release lock after critical section finishes

→ We don't have to worry too much about the details for implementing a Lock object, our processor & JVM does most of this work.

• Just an example to illustrate the logic behind synchronized.

→ When a thread for some reason, never releases a lock, and then other threads are never able to acquire it / access the methods.

→ If a thread dies in the middle of execution of the method (because it threw an exception).

• SOLUTION: put the method code inside a try-block, & put the .unlock() statement inside a subsequent finally block!

• Now, if the method throws an exception, it will still come back & execute the code in the finally-block (RECALL: catching an exception).

```
public void addOne() {
 lock.lock();
 try {
 value = getValue() + 1;
 } finally {
 lock.unlock();
 }
}
```

Is this the only situation where a deadlock could arise?

→ No. In fact, thread errors usually aren't the cause of deadlocks; it's usually a lot more complicated than this.

What is the usual/more likely cause of deadlocks in concurrent programs?

- Usually, we have a much more complex program where we have some **synchronized** method on one object that then calls a method (of another object) that is also **synchronized**
- For example, imagine **thread1** running inside a **synchronized** method of **object A**, called **method1()**, and this particular method happens to call **object B's synchronized method, method2()**
  - **thread1** has obtained the lock on **objA**
  - **thread1** is waiting on the lock on **objB** (so it can perform that method call)
- At the same time, **thread2** is currently inside of **method2()** (in **object B**), and **method2()** calls **A's method1()** inside its code.
  - **thread2** has obtained the lock on **objB**
  - **thread2** is waiting on the lock on **objA** (to perform the method call)
- Now, the threads are stuck... neither of them can continue because they are waiting on the other to release its lock... they are **deadlocked**.
- Especially in larger, more complex concurrent programs, it's hard to rationalize all of the paths existing in the program, and there are usually a lot of synchronized methods.
  - Because of this, deadlocks are one of the hardest parts of debugging concurrent programs - you often don't know why (where the deadlock has occurred).

Why is this deadlock scenario more likely to occur?

What are the **wait()** and **notify()** methods?

- methods defined by the Java **Object** class that are mechanisms for coordination between threads as they are running
- For a thread to call **wait()** or **notify()**, it must currently **own** the lock associated with the object
  - aka, the **wait()**/**notify()** call must be within a **synchronized** method's body (b/c that is when a thread owns a lock)
- Causes the current thread to "wait" - pause execution - until another thread calls either the **notify()** or **notifyAll()** method.
- Releases **one** waiting thread, as soon as the lock is available.
- Releases **all** waiting threads - each one resumes in turn as the lock becomes available.

What does **wait()** do?

What does **notify()** do?

What does **notifyAll()** do?

# Final Exam Review

## Unit 1:

### Java, Maven, Git

→ 2 steps to running Java code:

1) compile the source code from .java file to .class file (a diff. like language) → bytecode

human readable → JDK → .class  
.java code → binary  
binary → bytecode

2) execute the resulting bytecode

bytecode (.class) → Java Virtual Machine → executed!

↳ JVM is the universal "machine" that Java uses to be able to be a compiled lang that is also platform independent (Java's "big idea")

→ Interpreted languages: Farther from computer's level b/c need to be "interpreted" (parsed/translated)

First... therefore slower

Ex: python, javascript...  
newer languages

- langs that have a built-in code/program that interprets the code for execution

- platform independent: can run same source code on any machine

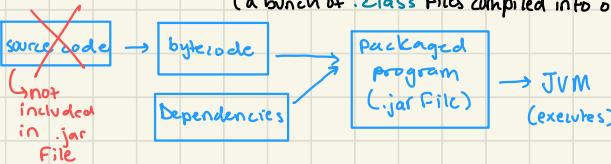
→ Compiled Languages: non-universal compiler that turns source code into machine executable code

- faster b/c speaks directly to computer

- platform dependent: compiling is done for the specific architecture used by the machine

Ex:  
C, C++, Rust...  
Older languages?

→ Dependencies: 3rd party libraries containing code that you use in your project, usually in .jar file format (a bunch of .class files compiled into one)



→ import statement: allows you to add a diff package to your class file so you can reference its contents easily

→ SKIPPED: "Java Build Process", "Git version control system" (Pages 5-9)

## Unit 2:

### Object-Oriented Programming

- basically the idea of collecting all the info & behavior for how a certain set of data is interpreted, maintained, & used into one "abstraction" — aka an object/a class — that can then be used to query properties, invoke behaviors, & save objects (specific set of data associated w/ one instance of the class) of that type.
  - Think of integer coordinates & integer lengths as the data... & "Triangle" as the abstraction/object
- **object**: a collection of named fields that represent info about it... each object is an instance of the class
- Parts of a class (an "abstraction"):
  - **FIELDS**: the pieces of info that collectively define the class object
    - int width, int height, & Color color for the SolidColorImage class from a06
  - **CONSTRUCTOR**: creates new instances of the class by filling the **Fields** with data values (usually passed to constructor by the user, as parameters)

```
public SolidColorImage (int h, int w, Color c){
 this.height = h;
 this.width = w;
 this.color = c; }
```
  - **INSTANCE METHODS**: behaviors/actions that can be performed on a specific instance of the class, & return an answer
- "static" = global; can be called globally; not specific to an instance but to the class as a whole.
  - any methods or variables/fields declared as **static** are referenced via the class name (!!), NOT the name of any specific instance.
  - SolidColorImage.getAssignmentNumber (for ex)
  - static is also the keyword used when creating methods in the main method (aka in a non-OO context)
- "final" Keyword : Fields marked **final** can't be reassigned (value can't change) AFTER constructor has given them their initial value
  - can still be instance-specific (doesn't have to be one, unchangeable var for the whole class — unless we also make it **static**) ... but for each instance, it can't change after being set
- Class specific versus instance specific:
  - **class fields/methods** : fields & methods NOT specific to any particular instance... one constant value that every instance has in common.
    - \* declared using **static** and **final** keywords!
  - private static final double EPSILON = 0.001;
- **instance fields/methods** : methods & fields specific to each instance... no static keyword
- **KEY TERMINOLOGY**: "class members" — All methods & Fields defined in a class (both "instance" and "class" ones)... but NOT the constructor.

## Unit 2:

### Encapsulation

→ **Principle 1:** shield an object's internals from the rest of the program (aka other .java files) in order to prevent instance fields from accidentally being changed, & to be able to refactor internal code w/o breaking external code.

**ENFORCED BY:** marking all class fields as **private**

→ **Principle 2:** Explicitly define "external" and "internal" behavior (which is like helper methods & etc.) in order to make objects easier to understand, maintain, use, and modify.

**ENFORCED BY:** defining an interface!

### Encapsulation Recipe:

1. make all **instance fields** **private**
2. Initialize instance fields with a **public constructor**
3. Add **getter & setter methods** to expose raw (private) field values
4. Carefully choose methods to expose as **public**
5. Make an **Interface** to clearly indicate which methods are exposed

(BECAUSE! an impl object CAN have extra methods (like helper methods) that aren't defined in the Interface... however, other ppl can't access those methods when creating objects of the **interface type** (aka "programming to the interface"). So we might as well make those 'extra' methods **private**.

### Access modifiers:

- **private** : only accessible inside the **class** they are created in
- **protected** : class & all of its subclasses can access it
- **default** : anyone in the same **package** has access
- **public** : anyone anywhere can access

→ **SKIPPED**: derived getters, setter validation, and all notes on interfaces, abstract methods in interfaces, etc.

## Unit 3:

### Inheritance

→ declare subclasses using the keyword "extends" ... public class Avocado extends IngredientImpl

→ subclass constructors: super( ) same parameter args taken by the parent class

```
public Avocado (String name, int amount, boolean isVegan) {
 super (name, amount, isVegan);
}
```

3

Avocado takes the parameter data given to it by the user, then calls super() & passes the data into there. super() calls the parent class constructor.

→ subclasses automatically inherit all class members (static & instance; private & public/protected etc.)

→ multiple inheritance (having more than 1 parent class) is ONLY allowed/possible for interfaces - not classes

- this is basically a "workaround" to the single-inheritance rule for classes, b/c a class can implement an interface which extends multiple interfaces (aka multiple object types)

### → TYPECASTING:

1. **upcasting** - taking a reference to an obj that is a subclass, & forcing it to be of its parent class type

- don't need to perform it using typecasting syntax; its implied & compiler knows its true
- checked @ compile time.

```
Student s1 = new Student();
Person p1 = s1;
```

remember, the "type" of the object is whichever class reference it is created as (aka the left side of the statement);

Image img = new ImageImpl();  
(img is of type Image)

2. **downcasting** - opposite of upcasting - Forcing obj to be a subclass type.

- needs to be performed using syntax:

```
Person p2 = new Student();
```

```
Student stu = (Student) p1;
```

- checked at runtime; prog throws exception if you made an illegal downcast.

### → Is-a relationships:

- all classes have is-a with the parent classes they extend (if they do)

Avocado is-a IngredientImpl

- Parent class objects only have is-a w/ a subclass object if they have been declared as that subclass obj.

Person p2 is-a Student ✓

Person p3 = new Person();

✗ p3 is-a Student → NO because we haven't specified what type of Person p3 is.

- all classes have is-a with the interfaces they implement.

IngredientImpl is-a Ingredient (the interface)

- all subclasses automatically have is-a with the interface(s) implemented by their parent class.

Avocado is-a Ingredient

→ Java's Object class is the "mother" parent class of ALL classes ... only has 2 methods, equals(Object) & toString()

↳ got 9 wrong on exam 1 b/c have to mention Object as a parent class!!

## Unit 3:

### Inheritance

How to Answer subtype polymorphism questions on the exam!

→ make a graph charting all the implementations and extensions



→ An object can only be instantiated as a certain type if the "type" has an is-a relationship with the "reference" (guaranteed).

`A obj = new C();` is NOT valid because "C is-a A" is not necessarily true... if you can't make a subclass Obj of parent class type?

- for questions asking about instantiation statements (like above), check to see if you can get from the type (right side of equal sign) to the reference (left side) by following a forward chain of arrows.

• `Student stu = new Person();` not allowed?

→ For problems about typecasting to a diff interface, don't think about the graph or is-a statements so much as the conceptual meaning of an interface:

`I2 obj = new F();`  
`IO obj2 = (IO) obj;`

is valid!

- F extends A, which extends D and implements I2, D extends C, C implements I1 (which extends IO)  
 so in summary: F extends A, D, C  
 F implements I2, I1, IO
- Since "F implements IO" that, by definition, means it provides impl for every single method defined/required by IO..
- this statement, an "IO" interface-type object doesn't care about any of the details of the object EXCEPT that it contains all necessary methods ... which obj does (since it is a type "F"), despite being a type "I2"

## Unit 4 :

### Reference vs Value Types

→ value type:

|        |       |       |         |                         |
|--------|-------|-------|---------|-------------------------|
| int    | short | byte  | char    | }] Java's 8 value types |
| double | long  | float | boolean |                         |

- items of these 8 types are stored **directly in memory** as their specified "value" (which is a string of 0s and 1s), in their specified location.
- b/c they are smaller pieces of data, they can be assigned their own "value" number-thingy ... obviously we can't store all objects this way b/c it would take up way too much space (for ex, an **int** is stored as a string of 0s & 1s that is small enough to only take up 4 bytes of memory)
- Everything else that isn't one of these 8 is an **object**, and thus stored as a **reference type**.

→ reference type:

- the value of the variable/object (aka the declared name of an obj instance) is not a string of numbers, but a **memory address** that is stored directly in memory.
- the memory address is a **reference to the object** that points to the location in the **heap** where the actual object's info is stored.

### Polymorphism

- The **is-a** stuff on previous page are all part of **type polymorphism** — when an interface and/or parent class has multiple implementations / subclasses.
- One type of polymorphism is when multiple methods in a class have the same name (aka several diff versions of the same method)
  - 2 versions: **Overriding** and **Overloading**

→ Method Overriding:

- When a subclass **opts to replace/rewrite the implementation of a method inherited from its parent class**.
- If you don't add **any code** to your subclass besides a constructor that calls `super(parameter args)`, it will still function completely!

```
public class Person {
 private String name;
 public Person (String name) {
 this.name = name;
 }
 public String getName() {
 return this.name;
 }
}
```

```
public class Professor extends Person {
 public Professor (String name) {
 super(name);
 }
}
```

Main:

```
Professor p1 = new Professor ("serrato");
System.out.println (p1.getName());
```

Output → serrato

## Unit 4 :

### Polymorphism

→ **Method Overriding:** When a subclass opts to replace/rewrite the implementation of a method inherited from its parent class.

• If you want to make a subclass-specific impl of one of the inherited methods :

→ use `@Override` compiler directive to tell compiler what ur doing

→ overridden methods can't access parent's private fields , so if you need that info, access it by calling the parent class' methods instead (if the field data is accessible that way) , using `super()`:

```
public class Professor extends Person {
 public Professor (String name) {
 super (name);
 }
 @Override
 public String getName () {
 String ans = "Dr. " + super.getName ();
 return ans;
 }
}
```

Main:

```
Professor p1 = new Professor ("serrato");
System.out.println (p1.getName());
```

Output → Dr. serrato

→ **Method Overloading:**

• providing multiple versions of the same method , but which differ /are distinguished by the parameters that they take in.

– MUST have diff parameters so that the call to each version of the method is distinct – so that compiler knows which exact method is being called

`rectangle.getArea (5);`      v.s.    `rectangle.getArea (5,3);`

• REQUIREMENTS:

→ must have either a diff number of parameters, or diff data type of the parameter args

→ must have same access modifier ( `public private protected default` )

→ must have same status as either `static` or not `static`

• NOTE: return type of the methods has nothing to do with overloading. The return types can be the same, or different.

→ Think about it – the whole issue of overloading is being careful not to confuse the compiler when calling an overloaded method. And when you call a method , you don't assume or know anything ab the return type.

```
public class Rectangle {
```

...

```
public int getArea (int width) {
```

```
 return (width * width);
```

```
public int getArea (int width, int height) {
```

```
 return (width * height);
```

## Unit 4 :

### Polymorphism

#### → Constructor Overloading:

- providing multiple constructors for an object that also have to have different type or amt of parameters.
- use constructor chaining with the `this()` keyword

```
public class Student extends Person {
 private String class;
 public Student (String name, String class) {
 super (name);
 this . class = class;
 }
 public Student (String name) {
 this (name, "freshman");
 }
}
```

3

calls the  
dg constructor  
(above)

#### → Final Keyword for methods & classes:

- methods marked final cannot be overridden by any subclass
- classes marked final cannot have any subclasses at all!

#### → Method Access in Polymorphism.

- When creating an object , you only have access to the methods defined for the reference of the object , not the object type
- Similarly, when "programming to the interface" (creating objects of an `InterfaceImpl` type but with the `Interface` as the reference) ... you cannot access any extra methods created in the impl which aren't defined in the interface.

### QUALIFICATION: Method Overriding

- If a subclass overrides one of its parents methods , then any object of the subclass type (whether or not the reference is to the subclass object or its parent object, interface etc.) will utilize the overridden implementation when the method is called.

A processor = new B();

int result = processor.process(5,6);

- object B overrides A's "process" method and in this problem, we would use B's implementation, despite the reference type of the object being A.

## Unit 5:

### Composition and Aggregation

→ program designs regarding how objects relate to the objects that they encapsulate.

#### → Aggregation:

- the encapsulated/internal objects are basically more independent... they have their own lives, utilities, and meaningful purpose outside of the aggregated object & are referenced elsewhere
  - the internal objects can exist independently outside of the agg (in a meaningful way; like it makes sense to have them)
  - the encapsulated objects are provided externally, usually through the param. args. of the agg class' constructor
  - basically the fact that the constructor's parameters aren't just primitive fields, but also actual class objects;
- ```
public Course (Room room, Professor professor, String name, int credits) {  
    ... //Assigning the arguments to private Room, Professor, String & int object fields }
```

→ Composition :

- the encapsulated objects don't really make sense outside of the composed class (not usually shared by other abstractions)
 - the encapsulated objects are created internally, rather than passed in by user
 - often, the constructor doesn't take any parameters
 - often, no setters or getters for these internal objects; they aren't meant to be exposed
 - the encapsulated objects' states & functions are thus only accessible to the composed abstraction
- *EXCEPTION: Dependency Injection

```
public class VehicleImpl {
```

```
    private Engine engine;
```

```
    private Wheel frontLeft;
```

```
    public VehicleImpl() {
```

```
        engine = new EngineImpl();
```

```
        frontLeft = new WheelImpl();
```

→ coupling: when classes reference each other by name, creating a dependency between them

• the more named references there are between class files, the more highly coupled the code is.

• coupling is okay between classes in the same package; otherwise it's a bad thing.

→ Dependency Injection:

- A way to support low/loosely coupled code in composition classes by "injecting" specific (already created) instances of class objects into the composition, rather than having it create them itself.
- dependency injection makes composition programs look more like aggregations b/c it kind of goes against their nature... aggregations already inherently support D.I. just by their design/definition.
- How to execute it: Inject the other class objects through setter methods (setter injection) or through constructor parameters (constructor injection)

Unit 5:

- When creating an abstraction for a class that wants to implement multiple interfaces, you can do this through inheritance (`class ABCImpl extends ABImpl` (which extends AImpl & implements B) implements C)
OR through composition - `class ABCImpl implements A, B, C` but encapsulates private A, B, and C objects that it delegates to when it comes to defining the methods mandated by the 3 interfaces.
- If the abstraction lends itself to natural hierarchy. Otherwise, and if/when given the choice, favor Composition!

Abstract Classes

- **abstract methods**: methods in a parent class marked with the keyword `abstract` and for which there is no coded implementation defining what it does
- Similar to how methods look in an interface - just the method signature;
`public abstract String getStatus();`
- Since it has no definition, every subclass of the class w/ the abstract method **MUST "override"** that method & define it itself
- **abstract classes**: marked by the `abstract` keyword, classes where you cannot make an object of that direct type
 - the class needs to have subclasses, can't be instantiated without specifying a subclass object type.
 - A variable can have the `abstract` class as its reference, but never as its object type

```
public abstract class Person {  
    ...  
  
    public abstract String getStatus();  
    ?
```

```
public class Student extends Person {  
    ...  
  
    @Override  
    public String getStatus() {  
        //some impl code here  
        ?
```

↓ Main:

```
Student avi = new Student(); - allowed  
Person avi = new Student(); - allowed  
Person avi = new Person(); - NOT allowed
```

- If at least one method in a class is `abstract` the whole class must also be marked an `abstract` class.
→ You can choose to make & mark parent classes `abstract` even if none of the methods are.

Unit 6:

Error Handling

→ **exceptions**: unexpected, unusual, or abnormal situations that arise during execution of a program.

→ Early error-handling strategies

1. **Global error codes**: a global variable where we store an int representing an "error code" whenever something goes wrong.

- declare a public static int @ top of a class
- Anytime code does smthn where an error could occur, we have to check the variable to see if it is still 0 or has changed

→ **DRAWBACKS TO EARLY METHODS:**

* Reliant on documentation (made by the programmer) explaining what each error code/return val means — this documentation needs to be well understood by others using the program

* (global error codes) if 2nd error occurs while 1st is being handled, there is nowhere to store the code

2. **Special return values**: we designate a special value that a method should return if it initially attempts to return some out-of-range value that it shouldn't have produced.

- void functions: should instead be int methods that return a number indicating the error status (like w/ global error codes)
- Other functions: something—or could have the function return null

* Programmer's responsibility to remember to check for errors at every potential spot (otherwise program could continue on unaffected & cause bigger problems later)

* (global error codes) have to clear out global var's value after each time an error is handled

∴ Modern Strategy: Exceptions

Unit 7:

Exceptions

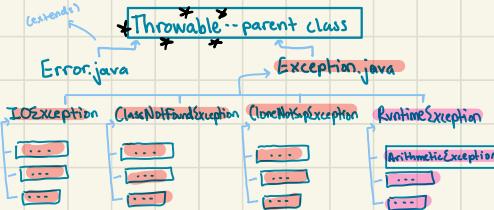
→ **exception handling**: Formal method for detecting & responding to errors; all languages provide a built-in mechanism for this.

• BENEFITS of exception handling (versus older strategies):

- consistent & extensive
- **Safer**: can assure that any code that needs to be executed will be (even if an error occurs)
- expressive (can encapsulate details about the error)

→ Exception handling in Java: Exception objects

- objects for each specific type of exception, that encapsulate details about the error that occurred - Java provides many built-in exception classes
- classified with inheritance:



→ "Error" represents externally caused, unrecoverable problems that generally shouldn't be caught/handled

* if program fully unwinds w/o error being handled, the program dies.

→ Catching an Exception

(like, in the `main` method...)
not some separate file

→ the "handling" aspect - providing the code to handle a thrown exception.

→ **try-blocks**: the block of code where we write the code/call the method that has possibility of an exception

→ **catch-blocks**: the block of code which contains the actual code handling the exception (how the prog responds to a given throw)

- usually multiple catch blocks, each one corresponding to a different type (class) of exception.

4) program jumps through the catch-blocks, looking for the (first) one that defines the same `Exception` class type (or a parent class of) the thrown exception.

5) executes the code inside the catch-block (ONLY the 1st "match" - doesn't look any further)

6) if no matching catch-block is found, program returns to "unwinding the stack" & repeating the process w/ the next method on the call frame.

→ Best Practices with Exceptions

1) Throw exceptions **EARLY** - as soon as you detect a wrong value

- **Defensive programming**

2) Be specific when throwing exceptions, & try to use a builtin type when possible

3) Catch exceptions **LATE** - let it "bubble up" to the level of the program where it will actually make sense

- Only catch it if you have some (programmatic) way to deal with the error

→ finally block: placed at the end of the sequence of catch-blocks and contains code that needs to execute no matter what -

- whether or not an exception was thrown

- whether or not it was handled by a catch-block

→ Throwing an Exception

• the "detection" aspect - signaling that smthn has gone wrong.

• Sequence of events:

1. exception object is created at the time that it is being thrown

`throw new RuntimeException ("no blah blah");`

exception class type

error message

2. Right after this line, the method/program stops executing & we start

"unwinding the stack" to look for a try-block.

3. Program unwinds & when it finds a method assoc. w/ a try-block, it goes to execute the subsequent catch-block which then handles the error.

Unit 7:

Exceptions

→ Checked vs unchecked exceptions:

Unchecked Exceptions ↗(on previous page)

- `RuntimeException` & all of its descendants; the `Error` class; and the `Throwable` class
- For errors caused internally within the program (ex: logic errors) that really "never should have happened" - eg programmer's fault
- Should only throw exceptions if we know how to handle the situation. May or may not need to address them in our code
- not subject to the "catch or specify" rule
- exception is thrown inside the method, but caught in the file where the method is being called.

```
(methodB() { if(x == 1) { throw new RuntimeException(); } } )
(main { try { methodB(); } catch(RuntimeException e) { ... } } )
```

→ "catch or specify errors"

- by specifying an exception in a method, we're basically "putting off" handling it
- we still have to catch it somewhere. 2 options:
 1. catch the exception in the `Main` method by calling it inside a `try`-block
 2. force the exception to continue "bubbling up" by having the `Main` method ALSO specify the exception (in its method signature);

```
public static void main(String[] args) throws FileNotFoundException { method(); }
```

PENNOMIC DEVICE:

Unchecked = `R`untime (&`Throwable`&`Error`)...

Unnecessary to "catch or specify"

Checked = everything else... must "catch or specify"

Checked Exceptions ↗(on previous page)

- All other `Exception` subclasses (as well as `Exception` itself)
- Responding to errors caused by factors outside the program's control.
 - "Our prog is responsible for always responding to these"
- Subject to the "catch or specify" rule.
- Exception must be caught (or specified) inside the method itself (not just the file where it is being called.)
- "catch or specify" rule: if a method contains code that might throw a checked exception, then the method must also EITHER:
 - catch the exception internally (with `try`-&`catch`-blocks)
 - do this iff the current method is the correct place to handle the error (and we know how to deal with it)
 - OR
 - Specify in the method signature that the checked exception might be thrown by the method:

```
public int methodC() throws FileNotFoundException { ... }
```

(best practice)
→ rule #3

- do this if the error needs to be dealt w/ at a higher level
- basically "instructs" the error to bubble up

Unit 8:

JUnit

Unit 9:

Iterator

→ behavioral design pattern

→ provides a way to sequentially access & loop through the elements of any given collection (like an ArrayList, Binary Tree, HashMap, etc.) without exposing the underlying implementation of the graph.

• the **Iterator** object doesn't know anything about the elements of the collection or what they mean

→ we create a new **Iterator** class/object for every particular kind of collection (like HashMap, LinkedList, etc.), and for every specific way that we want to loop through the collection (i.e. alphabetical, depth-first, breadth-first, etc.).

→ Key points about iterators:

- multiple iterators can traverse through a collection at the same time.
- the iterator pattern assumes that the collection will **not** be modified while the iterator is being actively used.

→ What an iterator object does:

- extracts the traversal behavior of a collection into a separate object (called an **iterator**)
- encapsulates all of the traversal details;
 - current position
 - elements remaining till end

→ the **Iterator<T> interface**: the interface for object classes that are "iterators" - objects that, for a given collection, encapsulate the details of how to loop through it.

• to be an iterator, must implement this method

• We create a specific **iterator** obj to define how to loop through a specific type of collection - therefore, we specify a data type **T** in a given **Iterator<T>** implementation.

Ex: `Iterator<String> iter = new Alphabetizer(data);` - Alphabetizer is an object class which implements the interface specifically for String objects.

• **boolean hasNext()**:

* figures out if there are still items left to visit, & returns false if not

• **T next()**:

* returns the next item in the collection (this is where you define the sorting logic you want to use)

* throws **NoSuchElementException** if no items remaining in collection

- the first thing that the **next()** impl should do is call **hasNext()**, and throw the exception if it returns false.

methods required by **Iterator<T>** interface

→ How to implement an **Iterator<T>** class:

• Requirements:

→ track progress through the collection

→ know which items have been seen, & which are coming up next.

→ manage the order of the items **WITHOUT** modifying the underlying collection & its order

Unit 9:

Iterator

Iterable<T>

- interface representing a class (usually one representing a collection) that is capable of creating and returning an Iterator object for its elements
- only defines 1 required method: `Iterator<T> iterator()`
- any class can implement `Iterable<T>` so long as they provide that method.
- All of Java's collection classes implement `Iterable<T>` (Map, List, Set, etc.)

Unit 10:

Decorator

→ allows us to extend or modify the implementation of an interface **W/O** subclassing / inheritance
→

Unit 13:

Graphical User Interfaces

- the original asynchronous programming model
- made up of UI components (widgets)
- AWT: old GUI library for Java
 - limited to just the UI components that were in common between all operating systems.
 - platform dependent
- JavaFX:
 - modern, well-known, widely used 3rd party GUI framework that is **platform independent**
 - inspired by web-application development
 - created with **responsive design** in mind.
- **Responsive Design**: enforcing a separation of content from style

Model - View - Controller

- a software design pattern used for structuring programs for user-interface applications.
- in effect, it **employs the Observer DP**.
- View/separate an application into 3 parts:
 1. **the state** of the application aka **how the app works** → (**the Model**)
 - the info that our app uses & manipulates
 - the logic/algorithmic stuff for data manipulation
 - the current status of the app's operations
 2. **the way the application looks** & is presented to the user, & how the user interacts with it → (**The Controller**)
 3. A means to translate the user interactions (#2) into manipulations of the underlying application state (#1)
- Big ideas of MVC:
 - **separate an applications UI code** from its **state management code**
 - have each of the **3 components** have their own well-defined interfaces & responsibilities.
 - **decouple the View from the Model** - useful bc then we can make diff views for diff devices or etc. (compatibility)
and just substitute them into the program.
 - **Model & View** don't even know that each other exist (ideally); both are **fully independent** & could be run on their own

→ The Model :

- stores the application state (all the stuff in #1)
- **Knows how the application works**, but **NOT how to show it to the user**
- is a **subject object** observed by either the View (classic MVC) or the controller (alternate MVC)
 - observed for state changes (bc that means aspects of the app have changed)

→ the Model:

- has 4 main responsibilities:

1. encapsulate the application state (in private fields)

private boolean[] lamps private Puzzle puzzle

2. expose methods for **accessing** the state (like getter methods)

isSolved() isLamp() isLampIllegal() getPuzzle getPuzzleLibraryIndex isCubeSatisfied()

• the observer (whether its View or Controller) will use these methods to update the UI

3. expose methods for **modifying** the states (setter methods)

addLamp() removeLamp() setActivePuzzleIndex() resetPuzzle()

• View/Controller will use these to reflect changes made by user interactions

4. notify its observers when any state has changed (aka after one of the above methods is called)

→ the View:

- Knows how the application looks, but not how it works ... everything in #2

- creates & displays the user interface using the current state data encapsulated in the Model

- has 3 main responsibilities:

1. create + display GUI

2. refresh/regenerate the UI whenever any application state changes occur

- does this by being an **observer object**, either **observing** the Model itself (classic MVC) or **observing** the Controller (which observes then the Model)

- when notified of a change, it is coded to then update the specific UI component that has to do w/ that change ... a tile on the Gridpane, the Label displaying the active puzzle index, etc.

3. To observe for user interactions & report them to the Controller

- does this by encapsulating a reference to the Controller (in the constructor)

→ The Controller:

- handles user interactions ... the "brains" of the operation

- translates user interaction events into commands for the Model

- must encapsulate reference to the Model.

Unit 14:

Concurrent Programming

- **sequential computing model**: normal computation where a series of computations are executed one at a time.
 - goes along with the **synchronous programming model** - where a task may be started (like calling a method), and the main program must pause execution & wait for the task to complete, before continuing on.
- **concurrent computing model**: series of computations executed over overlapping time periods. 2 ways to do this (unclear what the difference is):
 1. **asynchronous programming model**: model where a task can be started, but the main program continues on executing while waiting for it to complete, & later coordinates with the task
 2. "**parallel programming**": multiple tasks can simultaneously be executed on separate processing elements (cores).
 - * executed in Java using **threads**
- **threads**: an abstraction for executing a program, created so that the prog. can operate in more than 1 place at a time.
 - A Thread encapsulates 3 things:
 1. Instruction Pointer → Where we are in the program
 2. Call Stack → which methods are currently executing
 3. Memory → shared between all threads (unlike the above things)
 - Threads communicate with each other via coordination of their shared memory.

Implementing Threads in Java

main {

```
Runnable task1 =  
    () -> {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i);  
        }  
    };
```

lambda expression defining
Runnable's void run() method

```
Thread thread1 = new Thread(task1);  
thread1.start();  
}
```

- **Runnable**: defines objects that represent a task that can be performed ... "mini programs"
 - **run()** contains all the code we want executed for that task

- **Thread object**: represents a thread of execution ; takes a **Runnable** object as a parameter (providing the instructions on what the **Thread** should do)
 - call **.start()** method to start the thread - & thus begin concurrent execution.

- **.join()** method: called on a **Thread** object, and signals to the main method (where the thread was started) to pause its own execution UNTIL the thread is done executing all of its code.


```
thread1.join();
```

Unit 14 :

Thread Coordination in Java

- **race condition**: segment of concurrent code where the timing of the execution affects the result.
 - occurs when 2 or more threads are actively sharing memory:
 - manipulating or using the same object
 - retrieving state info from the same object
 - We don't want 2 threads to be manipulating the same object at the same time, because we can't predict which threads' modifications will end up as the final results.
 - **SOLUTION:** synchronized methods
- **synchronized keyword**: inside an object class, mark methods that engage in "reading from" or "writing to" the object as synchronized
 - public synchronized void addObj() { ... }
- methods marked as synchronized will not allow multiple threads to enter/execute them at the same time FOR A PARTICULAR OBJECT — the locks (exclusivity) is instance-specific, not class-specific.

Coding rules & tips

- don't use real numbers unless you have to — always opt for int
- static functions are global for the class they're in; they don't need to be at the top.
- "method" and "function" are synonymous
- NEVER compare two real numbers (`double`) values using double equal operator (`==`)
 - instead, take the absolute value of the difference between those 2 values, & compare to make sure that it is "less than" some small threshold value (which you define/decide) ... if it is, they can be considered equal.
↓
called the "epsilon bound"

→ `equals()` versus `=`

- `equals()` checks content equality ... are both of these objects identical?
- `==` checks reference equality ... are these objects the same obj in memory?
- EX:

```
String s = "hello";  
String y = "hello";  
String z = new String("hello");
```

- s == y ; → TRUE -- s and y refer to the same memory address in the "string constant pool"
 - s == z ; → FALSE -- z is a new String object
 - y == z ; → FALSE
 - s.equals(y) ; → TRUE
 - s.equals(z) ; → TRUE
 - y.equals(z) ; → TRUE
- [s, y, and z all have the same content / value]

- mnemonic device : `==` is "2 equal signs" which is INTENSE and OB...not just one, but 2...thus it must want the truest form of equality, aka that the 2 things being compared are actually the same object

`equals()` takes a value as its parameter, just wants to see if the object's content matches the value passed in

→ "dot equals" ... 'd' is right after 'c' ... think A,B,C content equality, Dot equals

→ stuff I skipped or should go back over my notes for:

Part 1 (beginning - Midterm 1)

→ Java Build Process

→ derived getters, setter validation (Unit 2)

→ notes on Interfaces & abstract methods in Interfaces (Unit 2)

→ meaning of final in various contexts

→ Java instance methods - always virtual

→