JUnit:

assertEquals();
assertNotEquals();
assertArrayEquals();
assertTrue();
assertFalse();

Functional VS Object-Oriented Design

Functional	OOD
• Define data first, then write functions to manipulate it. New data require new cases in the functions.	• Attach methods to data; data knows how to manipulate itself. • $$: Objects encapsulate both data and behaviors associated with that data.
• To add a new type: (e.g new web-page)	
• Need to add new struct which is fine. • Need to add new cond to the existing functions which is not good.	• Need to create a new object which is fine • No need to modify existing code.
• To add a new feature (e.g cite style):	
• Need to add a new function, no need to modify the existing functions which is great.	• Need to modify the existing code to add new method to each class type.
• Moral of the story: choose what you need.	

Reasoning:

- We can program with objects in nearly any programming language, though some languages make it smoother than others.

- It would be helpful the languages could find some kinds of bugs when design large systems, such as when an implementation or client doesn't conform to its interface. (In DrRacket, the interface to publication objects is informal, in a comment where the compiler doesn't see and can't check it.)
- So, we better off doing object-oriented programming in a language designed for it, with built-in notions of classes, messages, methods and interfaces.

- GOAL: How organizing our programs as cooperating objects can help us engineer software systems and components that are flexible, reusable, and maintainable.

Object-Oriented Design:

Good software:
Correctness & Efficiency (time & memory).
There is no clear metrics when making software design choices. With **experience** we develop intuition that distinguishes better designs from worse. One goal of this course is for you to gain a little more experience and grow a little more intuition.

It is important to note that lessons in design, similar to lessons in programming, are learned often through making mistakes. You should be ready for this: often you will discover design limitations only by trying them out. While you should always strive to come up with the *best* design before you implement it, be aware that it in the nature of design to evolve. No design is perfect for all situations, which means one can always pinpoint limitations in any design. The design process is about recognizing what you want, and then coming up with a design that satisfies your current and estimated future requirements, not every eventuality.

We are writing code without COMPLETELY understand it; just know enough to use it (e.g., how it executable is structured; like driving a car without knowing how to fix the engine).

Design Recipe:

———————

- By the end of the course, you will be expected to have a strong grasp of all of the design principles listed below, and be able to discuss and apply them in your own work. Some of them may contradict each other: With design, there is often no one right answer, but a matter of balancing tradeoffs and meeting the needs of the specific situation.

- Beware the spaghetti monster!
  - method code is long, unwieldy, and/or haphazardly designed. This is spaghetti code. Like trying to untangle spaghetti, understanding and debugging such code is unnecessarily time consuming and inefficient.
  - Break down the method into high level operations. Each of them become helpers. Make sure the method flows in how you call these high level helpers. Now you break down a helper into several operations, which become more helpers. Keep breaking down until you get manageable chunks of logic that you can implement. Finally, clean up if you think you have gone too far with your slicing and dicing, or if you think you have ended up with identical helpers. This top down approach introduces natural flow. Each chapter, section and subsection has a theme and a purpose. You can manage the chapter to chapter and section to section flow, because they are still all headings before you fill them in one by one.

- Write tests first, cover the range of situations, edge cases. Write code to be testable (avoid System.out); do not expose fields or add public methods just to allow for testing.

- Catch and handle/report errors as early as possible. Use Java compiler checks, enums, and final as your first line of defense, and runtime checks second.

- Don’t leave things in an inconsistent state for any substantive length of time.

- Use interface types over concrete classes wherever possible. Exception: immutable “value” objects (where all fields are final, or final classes with no interface.
    - Java allows a class implements multiple interfaces, if I want to add methods to a class, create an interface, and make the class implements it.

- Separate responsibilities: one class, one responsibility.

- Don’t duplicate code; reuse existing exceptions, classes, libraries, and designs.

- Open for extension, closed for modification: make changes without modifying existing code; write code to support later changes without modification.

- Extensibility: design to make likely later changes easier.

- Loose coupling over tight coupling. Write reusable components when possible.

- Use exceptions only for exceptional situations — not for flow control.

- Classes should never have public methods that are not in the interface (aside from constructor) — and you should not “fix” this by blindly adding such methods to the interface.

- Fields must always be private. Exception: constants (static final fields). Methods and classes should be as private as possible.

- When designing an interface, you can think ahead about the implementation but don’t let the implementation decide what methods should be included in the interface. Interface should be design only on how should the user use it.

- You can’t change an interface once it’s published.

- I can change it (physically, forcefully), but should I change it?

- In terms of design, at some point, I have to make a fundamental change. The goal is to make as less change as possible, or make the functioning time with no change as long as possible. After a week, you have to change your design fundamentally means your design is bad.

- If you override equals(), override hashCode(), and vice-versa.

- Use class hierarchies and dynamic dispatch over tagged classes, complex if/switch statements.

- Favor composition over inheritance

- Use class types over Strings.

- Before we start writing code:
  - First: Carefully figuring out an appropriate data representation of the relevant information we want to work with, limits on data;
  - Second: What operations it needs to support.

- Tests should be in the test directory, and in the default package. So it acts like a public user that can only see public stuff from you code.

Javadoc:

- Write Javadoc comments to all public classes and methods. The comment for a class should be at least two sentences, and provide information not already clear from its definition. A person reading your comments should understand the purpose, details of inputs and outputs without actually reading its code!

- We may omit purpose statements on these method implementations because their purposes are the same as the those written in the interface (and Javadoc knows to copy the documentation from there).
- Javadoc uses the first sentence of each Javadoc comment in the table of contents, so it should be clear and to the point.

- Each test case method must be preceded by the annotation @Test for JUnit to find and run it, must return void and must require no arguments.

UML class diagram (Unified Modeling Language):

- A dashed arrow with head means implements. (A solid arrow with an open head means extends).
- Italic type means the named entity is abstract; in particular, the methods in interface or abstract class.
- In each class, the first section is a list of fields and the second a list of methods.
- + means public; - means private; # means protected; underline means static;

Knowledge:

- Objects — data encapsulated with the code that knows how to operate on data. We use class to construct Objects.
- Clients of Objects — other code that uses them (interact with objects by sending them ***messages***. e.g Publication objects understand two kinds of messages, “apa” and mla”.)
- Method — the code that an object invokes in response to a message. (Helper methods should be private).
- Interface — the set of messages that it understands. To use Objects, we don’t need to know their classes, just interface.

- Final class means it cannot be extended.
- Final method means it cannot be overridden.
- Private field means client cannot access them directly. (As a rule of thumb, all instance variable should be private).
- Final field means it must be assigned exactly once, in the constructor, and once initialized its primitive value cannot be changed. If the field is reference type, final means you cannot reassign the name another reference.

- In a Java format string, the code `%d` means that the next parameter is an `int` . Writing `2` between `%` and `d` means to pad the number to be at least two characters long; by default padding is done with spaces, but `02` means to zero-pad the integer to make it two characters long. String.format("%d:%02d:%02d", hours, minutes, seconds);

- Time Converting:

```
public DurationImpl(int hours, int minutes, int seconds) {
    if (hours < 0 || minutes < 0 || seconds < 0) {
      throw new IllegalArgumentException("must be non-negative");
    }

    if (seconds > 59) {
        minutes += seconds / 60;
        seconds %= 60;
     }

      if (minutes > 59) {
         hours += minutes / 60;
         minutes %= 60;
       }

//after resetting the value of each param, then sign them to each field.
       this.seconds = seconds;
       this.minutes = minutes;
       this.hours = hours;
}

public DurationImpl(long inSeconds) {
    if (inSeconds < 0) {
      throw new IllegalArgumentException("must be non-negative");
    }

    this.seconds = (int) (inSeconds % 60);
    this.minutes = (int) (inSeconds / 60 % 60);
```

```
  this.hours = (int) (inSeconds / 3600);  // overflow...
}
```

- Long.compare(this, that);

- Interface polymorphism — the ability to use multiple classes that implement an interface with a client that depends only on the interface.

- In reality, often a happy client will not able to decouple itself completely from a specific implementation. This is because objects of the implementation would need to be created somewhere, mandating the mention of the class name in constructors. In this case, we would need to make a limited number of changes to the nearly happy client to update the constructors. We can isolate these changes to one or a few places so that these changes, although unavoidable, are manageable. (There is a way to design a client that avoids even calling a particular class's constructor, making the client completely decoupled from the class.)

- Console takes in everything as a string.

Abstract:

———————

- When we want to factor out redundancy between functions, we create a helper function and call it from the other functions.
- When we want to factor out redundancy between classes, we create a helper class and refer to it from the other classes. One way to use the helper class from the classes that share it is to have them extend it. Extension, also known as implementation inheritance, derives a new class from an older one by adding fields, adding methods, and replacing methods. Furthermore, we can design an

abstract class with extension in mind, leaving some of its methods missing for subclasses to define.

- Abstract classes are allowed to defined its owned methods. Factory methods (abstract) — concrete subclasses are required to define any abstract methods of their superclasses.

Static:

———

- In programming generally, **static** describes things that happen or are determined at compile time, dynamic describe things that happen or are determined at run time. In object-oriented programming, and in Java in particular, static means that some member — a field, method, or nested class — is part of its class which means static things are shared by all objects of the class. Whereas each object gets its own unique instance of non-static things. In a sense, static fields are Java's version of global variables, and like globals, they should be used sparingly, Public, static, non-final warrant extra suspicion.

- Static filed: ClassName.staticFieldName
  - Static fields are rare except for constants. A constant should be a **public static final** field whose contents are immutable, and its name should be in all caps.

- Static method: ClassName.staticMethodName
  Static methods are useful when they only work on data that is provided as their arguments, and do not require data stored in non-static fields. For example, the **Math** class has many static methods, because all of them operate only on what is provided to them as input.

    - The most common case for static methods are static factory methods, which produce objects of a class (and don't require that we already have an object to do so). It's also common to factor out implementation functionality into **private static** helper methods.

- Static class:
    - The concept of static classes only applies to nested classes (i.e. classes that are defined inside another class).
    - Nested classes are useful when one class has to define a class with the sole purpose of helping another class. Nesting this helper class inside the main class makes the latter more self-contained. An example could be an iterator defined over a list.
    - When nested class is static, it behaves like an ordinary class, and is not associated with a unique object of the outer class.
    - Furthermore, nested classes share the same private scope with their enclosing class. Meaning, they can see each other's private members.

```
//Case 1: Nested is a static class inside in Outer
//instantiating Nested
Outer.Nested nest = new Outer.Nested(…);

//Case 2: Nested is a non-static class inside Outer
//instantiating Nested requires instantiating Outer first
Outer outer = new Outer(...);
Outer.Nested nest = outer.new Nested(...); //nest is "linked" to object outer
```

- Use a static class when you want a helper class that's strongly associated with the enclosing class, especially when the helper doesn't make sense on its own. Nesting a helper class also allows the outer class to see its private members and vice versa, which can be helpful when the two classes are tightly coupled. For example, it makes sense to nest a class implementing an iterator for

a collection class inside the collection class, because the iterator probably doesn't make sense without the collection class, and it is often useful for the iterator to be able to see into the collection objects.

Array:

- When used to initialize a variable as part of its declaration everything before the curly braces can be omitted:
  int[ ] intArray = {2, 4, 6, 8};
- The **...** parameter: void setPlayers(String… newPlayers)

- **Aliasing:**
  - An array is represented as a reference to a chunk of memory, and the value of the array is the reference itself, not the chunk of memory. This means the re-assigning or passing an array results in aliasing — having more than one name for the same array.
- Both "==" and equals(Object) compare references rather than contents.
  - Use assertArrayEquals(…, …) to test contents equality.
  - Test myself: use Arrays.equals(Object[ ], Object[ ]) method which compares the elements of the arrays using their equals(Object). If the contents of the arrays are arrays, and still want to compare by the contents, use Arrays.deepEquals(Object[ ], Object[ ]).
- The index is not the position, it's the offset. Int arr = new int[7] arr contains the reference of the first int of the 7 (7 ints are stored together) .

Array vs List:

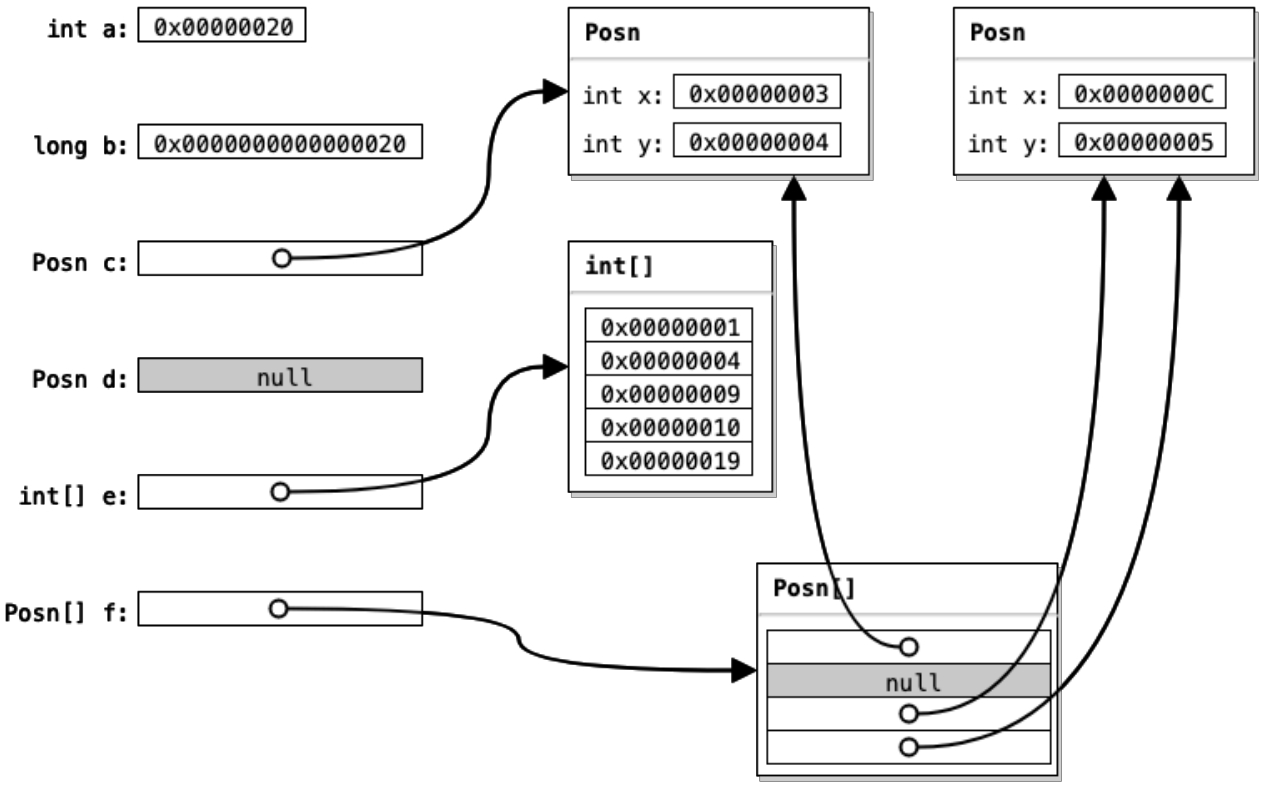
- Use an array to efficiently look up and update a sequence of values.
- Use a List when the length of the sequence needs to change.

- Since Lists can do everything arrays can, why would we ever use an array?
    - One reason would be if you know the length up front, and you want to guarantee that you can never accidentally change it. Another reason is for efficiency, since higher-level sequences such as ArrayList build on top of arrays with additional checking and indirection.

Characters:

- **char** type: letters(‘a’, ‘A’); digits(‘0’); punctuation(‘?’); whitespace(‘ ‘, ’\n’, ‘\t’, ‘\v’);

Primitive (built-in) type vs Reference type:



- In this diagram, there are six variables that none has a compound value (composed of multiple components). Each contains a single, simple value, which may be an immediate number, a reference to something else, or null.
- Variable a and b contain primitive numeric type **int** (four bytes) and **long** (eight bytes); these value is directly in the variable, with no references to anything else. The prefix 0x is for hexadecimal integer literals. Every bit in **int** or **long** represents a number, and there is no room for a distinguished **null** bit pattern; hence, **primitive types do not include null as a value.**

- The other four variables have reference types, of which there are two subdivisions, **object types and array types**.

- Variables c and d both have the **object type Posn**. Java variables cannot hold objects directly - objects are compound data structures — so instead they must hold a **reference** (the memory address of the object, like a pointer in C or C++).

- Variable c contains a reference to a Posn object. Note that fields of an object are a kind of variable, which means that they, too, can hold only primitives or references, not actual objects.

- Variable d currently does not hold a reference, so its value is null, which is a distinguished value that indicates the absence of a reference.

- Variable e and f have array types, which means that each hold a reference to an array (or null), not the array itself. Whereas e refers to an array of primitive int values, f refers to an array of Posn object references. Note that in the object, the third and fourth variable

contains references points to the same Posn objects — this is aliasing.

- The reasons why objects and arrays need to be accessed via references is that their types do not determine how much space they take up. In particular, an array type int[ ] does not determine the length of the array, and object type Posn could include subclasses with additional fields.

- In total, Java has eight primitive types that can be the immediate values of variables:

Type	Size	Description	Range
boolean	1 bit	truth value	True or false
byte	8 bits	Singed Integer	-2^7 to $2^7 - 1$
short	16 bits	Singed Integer	-2^{15} to $2^{15} -1$
char	16 bits	Unicode character	0 to $2^{16} - 1$
int	32 bits	Singed Integer	-2^{31} to $2^{31} - 1$
float	64 bits	Floating point number	
long	64 bits	Singed Integer	$- 2^{63}$ to $2^{63} - 1$
double	64 bits	Floating point number	

- They have known size, and each is a single value rather than some combination of values.
- Every primitive type in Java has a corresponding object type:
  - int has Integer (doc).
  - char has Character (doc).
  - Double has Double (doc).
  - Only int and char has different name, others with cap initial.

- Note that **short** cannot be null but **Short** can, **double** cannot be null but **Double** can, and so on.

Equality:

———————

- Value types vs Reference types (Things that contain an primitive value vs things that contains an arrow):
- For value types: just compare the values themselves
- For reference types:
  - We can see if the arrows point to exactly the same place, this check is what the == operator performs .
  - We can “follow the arrows” and recursively compare the data they refer to for sameness.

- Check two Objects are equal:
  - boolean equals(Object that) { return this == that; }
  - The default operation is simply intensional, physical equality. However we can override it.

- **Compatibility:** If x.equals(y), then x.hashCode( ) == y.hasCode( ) Contrapositively, if x.hashCode( ) != y.hashCode( ), then !x.equals(y)

- **Non-injectivity:** Just because x.hashCode( ) == y.hashCode( ) doesn’t imply that x.equals(y), Contrapositively, just because !x.equals(y) doesn’t mean that x.hashCode( ) != y.hashCode( ).

- To decide whether to override Object.equals(Object) and Object.hashCode( ). Never override one without the other.
  - The default implementations check the reference (==) and a hash function compatible with reference equality.

- For immutable value objects like DurationImpl, if we construct two objects representing the same length of time, those values are essentially equal. Thus it makes sense to override equals to define extensional equality.
- The static method Objects.hash(Object…) takes any number of arguments, each of which it hashes using that argument's hashCode( ) method, and then combining the results in a reasonable way.
- hashCode( ) to compute hash codes using the same values that are compared by equals. If equals(…) uses hours, minutes, seconds, then hashCode( ) returns Objects.hash(hours, minutes, seconds); If equals(…) uses seconds only, then hashCode( ) returns Long.hashCode(inSeconds( ));

- There is a utility class named Objects that includes a convenience method: int hash(Object … args) The actual mathematics of good hash functions are an interesting sub-domain of algorithms; for our purposes, its' enough to know Java has good built-in defaults that we can use as needed. **Of course, if our particular equality operation is more sophisticated than simply comparing fields, then this hashing approach won't work.**

Null:

———

- Null means "absence of an object". Since there is no object, using a variable that is **null** to call any methods results in a NullPointerException. This is more likely to happen when null is used to signify something (instead of nothing as it is supposed to). A classic example of such usage is null to signal the end of a linked list. An implementation using a sentinel (A last bogus object that

only means the end) avoids this pitfall. Thus in many cases, the use of null can be avoided.

- One of the few good use cases for null is when intending to create cyclic data. Here, use null to indicate that the cycle hasn't been formed; document well by what point the cycle should be formed, and check for it early and fail quickly if an unexpected null value appears.

- Nuanced advice: Use null only to mean "no object exists" and use sparingly only in this context.

Enum:

———

- Enums are used when we know all possible values at compile time. It represents a group of named constants.
- **Switch:**
  - Use **break** to end at the end of a case statement.
  - If case has a **return** statement, no need of **break**.
  - Every switch must have a **default** case as its final case. It may seem weird for enum, but enum values are objects of a given class type, and unfortunately **null** is legal value of that type as well.
  - Works on **primitive type**, **enums**, and "**String**"

Exception:

——————

- @throws IllegalArgumentException if the year is negative.
- checked and unchecked exception.
- More information about exceptions

Dimond Operator:

- Map<String, Integer> myMap = new HashMap<String,Integer>( );
  Java can infer the type parameters for us on the right-hand side of this variable declaration, so we can leave them out:
  Map<String, Integer> myMap = new HashMap<>( );

JUnit:

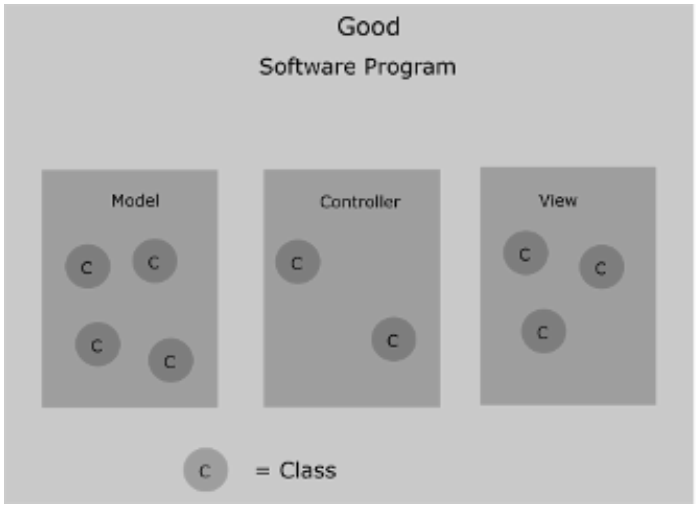
———————

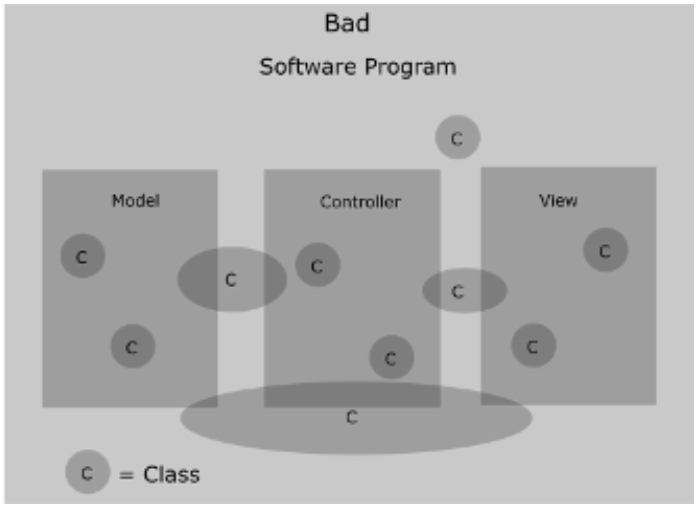
- It's a library.
- Import org.junit.*; //used to define @Test and @Before, etc.
- Import static org.junit.Assert.*; //used for assertEquals and assertTrue, etc.
- The assertEquals testing from compares its arguments using their **.equals** method. Keep this firmly in mind, as it is quite different from the tester library.

- We elaborate the annotation before the method with the expected exception's class:
  @Test(expected = IllegalArgumentException.class), and then simply invoke the constructor or method as normal.

- Note that testing will stop at the first exception thrown in each test method (short-circuits). If you want to test multiple exceptions, you much write multiple test methods.

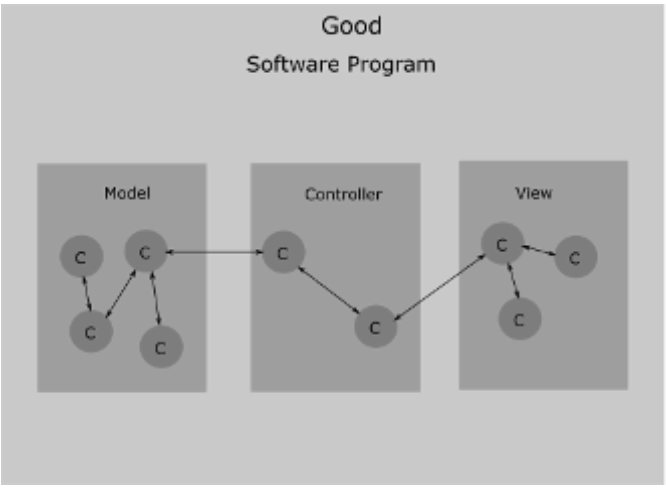
Model-View-Controller(MVC)

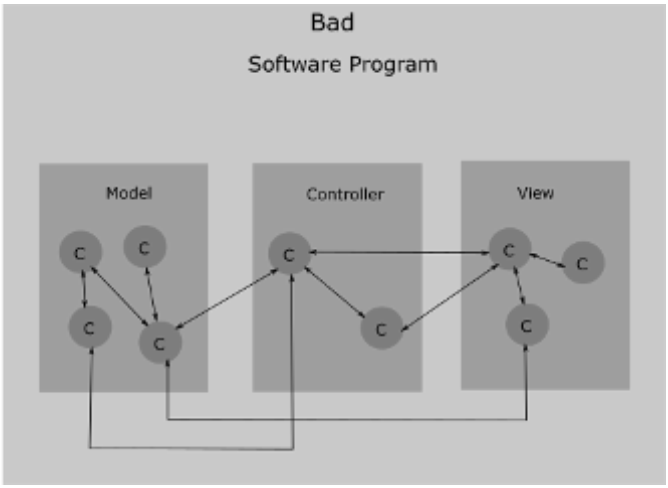




- Where do you start and structure you program if you want to implement a graphical game?
- A common object-oriented technique is **mvc** pattern which separates the program into three distinct but cooperating components:
  - The model: represents the state of the game internally and (often) enforces its rules
  - The view: knows how to display the game interface to the user
  - The controller: takes input from the user and decides what to do
- Each class should fall in exactly one of the model, view or controller.
- Since the model, view and controller have separate functions, access to each other is also restricted. Typically the model and the view cannot directly access each other, and the controller communicates with both. In many programs the view cannot ask the controller for data: the controller decides when to provide data

to it. In doing so, the MVC design promotes low coupling between groups of classes.





- Controller only interacts with one Object in the Model, and one Object in the view. So think Model and View as a whole.

- Another way to think about the merits of MVC is what each part does not do:
- Model:
  - Does: implements all the functionality.
  - Doesn't: does not care about which functionality is used, when, how results are shown to the user.
- Controller:
  - Does: takes user inputs, tells model what to do and view what to display
  - Doesn't: does not care how model implements functionality, does not care how the screen is laid out to display results.
- View:
  - Does: Display results to user
  - Doesn't: does not care how the results were produced, when to respond to user action

Design-Data:

1:it's better to make bad states or operations inexpressible than to catch them later.

2: when we want different behaviors, we should use different classes rather than a bunch of conditionals.

Builder: hide the constructor from client. Implementing a method for each parameter to set it individually.

- In order to enforce the rules and prevent client code from corrupting the state of the grid, we need to ensure that all changes to the state happen via our model class's public methods, which will enforce the rules. Thus, we cannot merely return a reference of our internal mutable representation of the grid to the client. However, we can safely return a copy of the grid (which need not be the same type as what we use internally)

View:

Knows how to display the game interface to user

So far we've worked on designing ***models*** to represent the data relevant to a problem domain, in a form that encapsulates the data behind an interface that clients can use without having to know any implementation details. The model is responsible for ensuring that it can't get stuck in a bogus or invalid state, and expose whatever appropriate observations and operations are needed while still preserving this integrity constraint.

We've also worked on simple synchronous ***controllers*** that allow users to interact with a model, in a form that encapsulates the user interactions and can provide feedback to users without having to redundantly ensure any integrity constrains. Moreover, controllers can be customized or enhanced without needing to change the model, making the model more convenient to use without making it any more complex.

Now, Views are renderings of the data in the model, and can be as simple as printing debug output to a console, as complex as fancy graphical user interfaces, or anything in between.

- **Choices for players of TicTacToe:**
  - ? nextPlayer( );
  - Type **char**, with ‘x’ for X and ‘y’ for Y
  - Class **String**, with “x” for X and “y” for Y
  - Type **boolean**, with true for X and false for Y
  - An **enumeration** defined as enum Player {X, Y}

  - Using type **char** is a poor design choice because the char type has many other values that don’t stand for valid players. Java’s type system will not help us in preventing nextPlayer from returning meaningless values. More to the point, the char type is good for representing textual characters, and abusing it for other meanings is unstylish.

  - Using a **String** is even worse, because it has all the same drawbacks as using a char, except it has even more possibilities, including null, and longer strings. *(Important guideline you should always follow: Strings are for representing textual information, where the possible values are many or unlimited and not known ahead of time. If you use string to represent some small set of values, or for any kind of internal API communication that is never presented to the user, you are doing it wrong.)*

  - The **boolean** idea solves the main problems that we saw with char and String: there are exactly two Boolean values, and we need two values. However, boolean could be confusing, because which player is true and which false is not obvious. In general, we should use booleans only when the value we are trying to represent is a truth value.

- Finally, we come to the enumeration, which declares a new class Player with exactly two values: Player.X and Player.Y This expresses very clearly what the possibilities are and what they mean. One caveat with enumerations, though: switch statements over enumerations always require a default case, even if every current possibility appears in some other case. This is because Java assumes an open world, in which more values can be added to an enumeration at a later time.

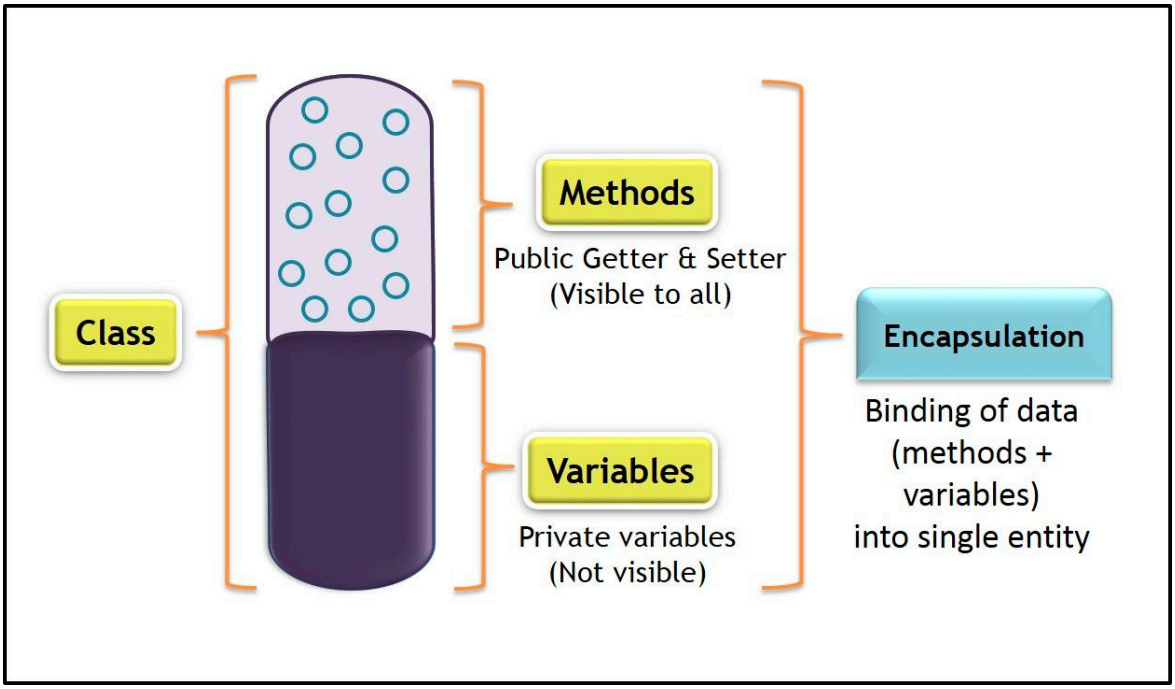
- **Try-Catch Exception:**

```
public void testVenderBoard( ) {
        try { msv.renderBoard( ); }
        catch (Exception IOException) {
                //the test should fail here. Or do any thing I want here.
                fail("Not valid Input!")
        }
}
```

OR:

```
Public void testVenderBoard() throws IOException {
        msv.renderBoard( );
}
```

- When calling a method inside another, either try it and catch the Exception, or the outer method should throws the Exception.
- You only catch what is thrown, not what I want to throw, but when I catch what is thrown, in the catch block, I can throw what I want to throw.



Encapsulation:

- List<List<Integer>> columns:
  - Integer because list takes in an object, and we can use null.
  - Using List instead of specific class such as ArrayList to declare.
    - So you don't have to change the fields, but just the code when you are using the arrayList when you need to change.

- Map<String, Object> properties:
  - can represent any class I want, the most flexible representation ever. Any field has a name and value.
  - Now we're programming in Python.

- Gain vs Lose when put everything in Map:
  - We gained a lot of flexibility
  - We lost our expression of intent, the clear meaning of the several named fields, the ability to control the shape of our data.

- To prohibit bad things: using the correct language features; or by careful programming.

- Access modifier:

Modifier	class	package	subclass	world
`private`	✓	✗	✗	✗
default	✓	✓	✗	✗
`protected`	✓	✓	✓	✗
`public`	✓	✓	✓	✓

- When multiple classes within the same package are cooperating in some close way such that it doesn't make sense for them to communicate via interfaces, you can use the default access level.

- **Class Invariant:** A logical statement about the instantaneous state of an object that is ensured by the constructors and preserved by the methods.
  - A logical statement is a claim that is true or false.
  - The instantaneous state of an object is the combination of values of all its fields at some point in time.
  - The invariant is ensured by constructors in the sense that whenever a public constructor returns, the logical statement holds.
  - Preserving the logical statement means that the method doesn't introduce nonsense — instead, we know that if given an object in a good state then it will leave the object in a good state as well.

- In order to apply class invariant, we need to determine what invariants we have, and then check the code to make sure that's true.
- Class invariants enable a form of reasoning called **rely-guarantee.** The idea is that components **rely** on some things being true when they're invoked, and they guarantee some things being true upon return. In particular:
  - If the constructor ensures some property
  - Every method (or means by which the client can mutate the object) preserves the property.
  - Then every public method, on entry, can rely on the property.
- In this way, class invariants allow you to rule out object representations that you don't want.

- A **structural invariant**, on **heaps**, the fullness property, that ensured the tree with n items was always as short as it could possibly be, which in turn ensured O(log n) performance. It also came in handy when we looked at clever data representations of heaps using arrays.
- A **logical invariant**, the heap-ordering property, that ensured the largest values were always near the root of the tree and that all subtrees were themselves heaps.

- Class invariants are similar in spirit, but a bit different in scope. They focus more on making all the methods of a single class work properly in concert, so that other parts of the program can rely on the property being true. When a single class implements an entire data type, class invariants and logical invariants become largely the same thing, but it's rare for a data type to be implemented by just one class!

- Anonymous Class: Anonymous Classes
- Lambda Expression: Lambda Doc

Inheritance vs composition:

————————————-

Inheritance: (tight coupling)
derived classes reuse methods in baseclass; implements / extends.

- A circular problem may happen when a public method calls another public method, because the extended sub-class may override both of them and changing one affects the other. Rule of thumb, a public class should only calls a private helper method.

Composition:
Make a class final to prevent someone extends this class, and reuse this class via composition: stores a class as a field which means use an object of the class not class itself. Furthermore, coupling with an interface is better with a class because interface has no implementation thus pose less problems.

- have a private field of that specific class(coupling).
- have an interface as a field and let user pass in a specific class (louse coupling)

Design pattern: functional object, visitor, builder, mock object, factory method,
Builder:

————

Mock Object:

———————

Factory Method:

———————

- interface —> abstract class —> many other implementations.
- Duration —> AbstractDuration —> HMSDuration —> CompactDuration
- type one: inside the class, wants the factory class in abstract class.
- public Duration fromSeconds(long inSeconds)
-
- Can return several kinds of Object so the return type is the Interface.
- Return a specific object at runtime
-
- *type two: outside the abstract class*, but cannot stand alone, so inside another class. This class contains the same factory method but takes in an additional parameter to specify which class to create. I can use Lambda, but if I need this in many places, it makes sense to keep it in one class.
- class DurationCreate {
    public static Duration fromSeconds(long inSeconds, String/Enum) {
}
}
-
- only Abstract the common part, if I find I have to change a lot in subclass, it shouldn't extend the base class.
-
- move on immediately if stuck
- open two tab for accessing notes
- constantly copy paste from IntelliJ to Canvas
- just normal comments
-

Image:

———

- a sequence of pixels; each pixel has a position in the image (row, column) and a color;
- PPM: Portable Pixel Map, an image extension / type
- RGB: components aka channels; If 8 bits are used per channel, this creates the (traditionally-termed) 24-bit image (0-255). Some formats support transparency (e.g. the PNG): this is done by adding a fourth component. black(0,0,0); white(1,1,1); Red(1,0,0); Green(0,1,0); blue(0,0,1); yellow(1,1,0)
- greyscale image: when RGB have the same value; it measures “brightness” or “intensity” for each pixel in three ways:
  - Value: the maximum value of the three components
  - Intensity: the average of the three components
  - Luma: the weighted sum 0.212r+0.7152g+0.0722b

-
-
-
-
-
-
-
-
-
-
- d