# CSC207 Notes

Jenci Wei

Fall 2021

## Contents

## 1 Introduction to Java

In Java, no code can exist outside of a class

- No functions, only methods
- Use curly braces to indicate what code is inside of what

When we execute a program in Java, we actually execute a class

- The `main` method is executed
- E.g.

```java
class Hello {
    public static void main(String[] args) {
        // The method body will go here.
    }
}
```

Printing things: `System.out.println`

- E.g.

```java
        System.out.println(7 + 5);
```

- Semi-colon marks the end of a statement

In Java, every value has a type, and so does every *variable*

- Must specify type before assigning a value to the variable
- Type of a variable can never change
- E.g.

```java
        int i;
```

  and

```java
        int i = 4;
```

- Default value for int is `0`, default value for object is `null`

Java keeps track of 4 things associated with each variable:

1. Variable's name
2. Variable's type
3. Memory space used
4. Value of variable (the only one that can change)

Potential errors

- Did not declare variable
- Assign value of the wrong type

- Declare a variable using a pre-existing name

Java types

- `int`
- `String`
- `boolean`
- `double`
- etc.

References vs. primitives

- `String` is a **reference** type
  - A Java variable cannot hold a `String` value directly inside itself, it can only hold a reference to an object of type `String`
- `int` is a **primitive** type
  - A Java variable can hold an `int` value directly inside itself
- Primitive types all begin with a *lowercase* letter
- Reference types all begin with an *uppercase* letter

Three areas in memory:

1. Call stack - where we keep track of the method that is currently running
   - We have a "stack frame" for each method of the class (which are currently running), an inside we have the variables
   - Reference types have the ids associated with the variable name
2. Object space - where objects are stored
   - We have ids of reference types associated with their values here
3. Static space - where static members of a class are stored

Java has a class `String` that represents sequences of characters

- To create a new string object:

```
        String s = new String(‘‘Hello’’)
```

- Use **double quotes** for `String` literals in Java
- Shortut for creating string objects:

```
        String s2 = "bye";
```

- Strings are immutable in Java

`StringBuilder`: mutable `String`s

- A `StringBuilder` represents a sequence of characters

- A `StringBuilder` object is **mutable**
- Only way to create a new `StringBuilder` object:

```
    StringBuilder sb = new StringBuilder(‘‘ban’’);
```

`char` is a primitive type capable of holding a single character

- E.g.

```
    char c = ’x’;
```

- Use single quotes for literal value of type `char`

Mutating strings vs. new strings

- Use `StringBuilders` when we want to mutate the content often as constructing a new object is slower

We can create an instance of a class using the keyword `new`

- E.g.

```
    StringBuilder name = new StringBuilder("abcd");
```

- In brackets, we provide arguments for the constructor
- We can use the newly constructed object directly, i.e.

```
    System.out.println(new StringBuilder("A").append("B"));
```

We call an instance method via an instance

- E.g.

```
    String band = "Arcade Fire";
    int size = band.length();
```

Some methods are associated with the class as a whole, they are called **static methods** since they are defined using the keyword `static`

- We access a static method via the class name, e.g.

```
    double x = Math.cos(48);
```

How we access data members i analogous to how we access methods

- If a class has an instance or class variable (declared with the keyword `static`), it can be referred to via an instance variable or the class name, respectively

If we know that we don’t need an object anymore, we can drop a reference by setting the variable holding the reference to `null`

- This may improve performance, but makes code messy

To declare an array, we must say that the type is array (using square brackets) and say what type each element of the array will be

- E.g.

```java
int[] numbers;
```

- Arrays are reference types
- To construct an array (of length 5):

```java
numbers = new int[5];
```

- To construct an array with an initializer:

```java
numbers = {1, 2, 4, 8, 16, 32, 64, 128};
```

- Java arrays do **not** offer slicing and do not permit negative indices
- We can have an array where not every element have the same type using inheritance:

```java
Object[] arr = new Object[5];
```

  But we can only access its element as `Objects`; to have our desired types, we have to **cast** the `Object` to, e.g., a `String`:

```java
String s = (String) arr[1];
```

We can create arrays with multiple dimensions

- E.g.

```java
int[][] table;
table = new int[50][3];
```

- We can also create irregularly dimensioned arrays:

```java
int[][] irregular;
irregular = new int[3][];
irregular[0] = new int[6];
irregular[1] = new int[99];
```

Aliases

- With references, we can create aliases:

```java
String name = new String("Justin Trudeau");
String primeMinister = name;
```

- With primitives, we cannot create aliases (since no references are created)
- If the object is immutable, changing it results in the creation of a new object, thus the previously-aliased reference would still refer to the same object
- We can make copies in order to avoid side effects of aliasing

- **Shallow copy**: copying the reference stored at the first level; **deep copy**: making a copy of a variable at *every* level
- In Java, modifying the list passed as the parameter would modify the original, too

When we call a method:

1. A new stack frame is pushed onto the stack
2. Each parameter is defined in that stack frame
3. The value contained in each argument is assigned to its corresponding parameter
4. Method body is executed
5. When the method returns, the stack frame for that method call is popped from the stack and all the variables defined in it (i.e. both parameters and local variables disappear)

- When we pass a reference to a mutable object in a method, the object may be mutated
- When we pass a reference to an immuatble object in a method, the object outside of the method will stay the same regardless of what we do

## 2 Classes in Java

Two kinds of variablese we can declare for classes:

1. **Instance variables**
    - Every instance of the class contains its own instance of these variables
2. **Class variables**
    - Aka. **static variables**
    - All instances of a class share a *single* instance of these variables
    - Updating this variable in one instance will reflect across every instance of the class

Attributes and methods can be either **public** or **private**

- Use `public` or `private` keywords to make the distinction
- Privatre variables *cannot* be accessed from outside of a class
- Also a `protected` keyword, where the attribute or method is accessible to the entire package and a class' subclasses, but not to everything else

Constructors

- We can define as many constructors as we want as long as method signatures are different
- E.g.

```java
public Something(String name, int size) {
    this.name = name;
    this.size = size;
}
```

- Can create additional constructors, e.g.

```java
public Something(String name) {
    this(name, 1);
}
```

When we define multiple constructors, we are **overloading** the constructor

- Can be done to any method

We can re-define a method from a parent class in order to override it

- Must include the `@Override` annotation

`equals` method

- Different from `==`, which checks for identity equality
- `equals` also defaults to identity equality, but can be overriden

Whenever we override the equals method, we often need to override the hashCode method

- If two objects are equal (according to the equals method), they have the same hashCode

Class (static) methods

- Use the `static` keyword to declare that a method is a class method
- Called by prefixing it with the class name
- A class method **cannot** access an instance variable or call an instance method directly because there is no "this" when we are in a class method

The `sort` method requires that all elements in the array must implement the `Comparable` interface

- Objects we are sorting myst be able to compare across their classes

To make instances of a class comparable:

1. Implement the `compareTo0` method
2. Change class declaration to indicate that we have fulfilled the requirements of implementing the `Comparable` interface:

   ```
   class someClass implements Comparable<someClass> {
   ```

   - `Comparable<someClass>` indicates that an instance of this class can be compared to any other instance of `someClass`

To enable different kinds of comparisons, use a "comparator" class (i.e. that implements the `Comparator` interface)

- E.g.

  ```
  class firstComparator implements Comparator<someClass> {}
  ```

- To sort, pass the `Comparator` as a second argument

## 3 Relationships Between Classes

In Java, use `abstract` keyword to signify that no instance of the class should be created

When we override a parent class's method, we include an `@Override` annotation

In cases where we wasn't to define a property of a class, we can use interfaces

- Interfaces have no implementations, only method signatures
- Can implement many interfaces, but can only extend 1 class

Use `super` to refer to methods in the parent class

- E.g. `super()` for the constructor, `super.method()` for the Method

When extending another class, Java *requires* a call to a superclass's constructor to be made in the subclass's constructor

- E.g.

```
class Child extends Parent {
    ...
    public Child() {
        super(...);
        ...
    }
}
```

Casting: changing the type of an object to another

## 4 Assorted Topics in Java

Shadowing

- When we call the method of the following code:

```java
public class ShadowExample {
    private int shadowedVariable = 10;

    public void shadowingMethod() {
        int shadowedVariable = 20;
        System.out.println(shadowedVariable);
        System.out.println(this.shadowedVariable);
    }
}
```

  20 and then 10 will be printed

- Use `this` to refer to the class variable

Array Copy

- Java arrays have a `clone` method
- To make a deep copy of a nested array, we need to `clone()` all inner arrays

Autoboxing

- **Autoboxing**: a conversion that the Java compiler makes automatically between primitive types and their corresopnding object wrapper class (and vice versa)
- E.g. `int` and `Integer`

Generics

- If we have an `Object` class, we would often need to cast it before using
- We can specify types, e.g. `ArrayList<Integer>`
- To use generics:
  - Add `<T>` to class declaration, e.g. `pulic class MyItem<T>`
  - Replace `Object` with `T`
  - When we create an instance of `MyItem`, we can specify that `T` should be e.g. `Integer`

  Other ways to use generics:

  - `MyItem<T1, T2, ...>` for many generic types
  - `MyItem<T extends Integer>` to enforce that `T` must be `Integer` or a subclass of it

## 5 Version Control

Version control

- Master repository of files exists on a server
- People "clone" the repo for local copy
- People "push" their changes *to* the master repo, and "pull" other people's changes *from* the master repo
- Repo keeps track of every change - people can revert to older versions

Purposes

- Backup and restore
- Synchronization
- Short/long term undo
- Track changes
- Sandboxing (trying something without messing up the code)
- Branching and merging

Remote repository

- Repo that lives on another server
- Called *origin*
- "clone" command makes a copy of the remote repo on local machine
  - `git clone <url>`

Local repository

- When we "commit" a change to a local repo, we first need to "stage" the changes

Staging changes

- `git add` marks file as being part of the current change
- Run `gir add` to add the changes to the next commit

`git status` - four states

1. **Untracked** - never run a git command on the file
2. **Tracked** - committed
3. **Staged** - `git add` has been used to add changes in this file to the next commit
4. **Dirty/Modified** - the file has changes that haven't been staged

Basic workflow

1. Start a project by `git clone <url>`
2. Make changes to files and add new files
3. `git status` to see what has been changed

4. `git add file1 file2`
5. `git commit -m "meaningful commit message"` saves changes to local repo
6. `git push` pushes changes to the remote repo

Branch and merge workflow

1. Create a **branch** to avoid having to constantly resolve conflicts
2. On the branch, work on a new feature and submit **pull request**
3. Other team members review the pull request
4. Resolve conflicts and merge

## 6 CRC Cards

CRC

1. **Class**
   - An object-oriented class name
   - Includes information about super- and sub-classes
2. **Responsibility**
   - What information this class stores
   - What this class does
   - The behaviour for which an object is accountable
3. **Collaboration**
   - Relationship to other classes
   - Which other classes this class uses

Creating a CRC model

1. Identify **classes** (nouns)
2. Add **responsibilities** (verbs)
3. Identify other classes that this class needs to talk to in order to fulfill its responsibilities (i.e. **collaborators**)

Scenario walk-through

1. Start with the initial input for scenario
2. Find a class that has responsibility for responding to that input
3. Trace through the collaborations of each class that participates in satisfying that responsibility

# 7 Test-Driven Development

Unit testing

- Want to fully test each unit
- In Java, a unit is often a method

Assertion

- Single-outcome assertions
  - `fail`, `fail(msg)`
- Stated outcome assertions
  - `assertNotNull(object)`, `assertNotNull(msg, object)`
  - `assertTrue(booleanEx)`, `assertTrue(msg, booleanEx)`
- Equality assertions
  - `assertEqual(exp, act)`, `assertEqual(msg, exp, act)`
- Fuzzy equality assertions
  - `assertEqual(msg, expected, actual, tolerance)`

Potential results

1. **Pass** - test produced the expected outcome
2. **Fail** - test ran but produced an incorrect outcome
3. **Error** - test ran but produced an incorrect behaviour

Setup and teardown

- **Setup** - single method annotated with `@Before`, run before every test method
- **Teardown** - single method annotated with `@After`, run after every test method
- `@BeforeClass/@Afterclass` run once before/after all test methods in that test class are executed
- Setup and teardown methods are used to avoid repetition (i.e. to create/destroy data structures required for more than 1 test method)

Unit testing pattern

- Lots of small, independent tests
- Report passes, failures and errors
- Setup and teardown shared across tests
- Aggregation (combine tests into test suites)

Selecting test cases

- Test for success
  - General cases, boundary cases
  - 0, 1, more

  - Odd, even
  - Beginning, middle, end
- Check for data structure consistency (representation invariants)
- Test for atypical behaviour
  - Does it handle invalid input?
  - Does it throw the exceptions it is supposed to?

Testing guidelines

- $\geq 1$ test class per class being tested
- $\geq 1$ test method per method being tested
- Name test methods `testMethodNameDescription`

Design for testability

- Write methods that do a single task
- Separate input, computation, and output
- Modularity

Testing code with exceptions

- Method 1: `@Test(expected=IndexOutOfBoundsException.class)`
- Method 2:

```
try {
    ...
    fail("IndexOutOfBoundsException not thrown when ...");
} catch (IndexOutOfBoundsException e) {

}
```

Testing code with inheritance

- Can have a ‘parent test class’ and ‘child test class’ corresponding the parent class and child class

## 8 SOLID Design Principles

SOLID

1. **S**ingle responsibility principle (SRP)
2. **O**pen/closed principle (OCP)
3. **L**iskov substitution principle (LSP)
4. **I**nterface segregation principle (ISP)
5. **D**ependeny inversion principle (DIP)

Single responsibility principle

- Every class should have a *single* responsibility
- Equivalently, a class should only have *one* reason to change

Open/closed principle

- Software entities (i.e. classes, modules, functions) should be *open for extension, but closed for modification*
- Adding new features *not* by modifying the original class, but rather by *extending* it and adding new behaviours

Liskov substitution principle

- If `S` is a subtype of `T`, then objects of type `S` may be substituted for objects of type `T`, without altering any of the desired properties of the program
- In Java, "`S` is a subtype of `T`" means that `S` is a child class of `T`, or `S` *implements* interface `T`
- Related to OCP since subclasses should only *extend* behaviours, not modify or remove them

Interface segregation principle

- No client should be forced to implement irrelevant methods of an interface
- Users should not end up depending on things they don't need
- Better to have lots of small, specific interfaces than fewer larger ones
- Makes it easier to extend and modify the design

Dependency inversion principle

- The approach of having high-level classes to depend on low-level classes is not flexible
  - If we need to replace a low-level class, the logic in the high-level class will need to be replaced
- Introduce an **abstraction layer** between low-level classes and high-level classes
- Goal is to *decouple* the system so that we can change individual pieces without having to change anything more
- Two aspects to the DIP:
  1. High-level modules should not depend on low-level modules, both should depend on abstractions
  2. Abstractions should not depend upon details, details should depend upon abstractions

# 9 Clean Architecture

Architecture

- Divide the system into logical pieces and specifying how those pieces communicate with each other
- Goal is to facilitate the *development*, *deployment*, *operation*, and *maintenance* of the software system

Policy and level

- A computer system is a detailed description of the **policy** by which inputs are transformed into outputs
- Want to separate and group policies as appropriate
- A policy has an associated *level*
- **Level** - the distance from inputs and outputs
  - Higher level policies are farther from the inputs and outputs
  - Lowest level are those managing inputs and outputs
- In Clean Architecture, *entities* are the highest-level policies

Entities

- Objects that represent variables and methods

Use case

- A description of the way that an automated system affects entities
- Use cases manipulate entities
- Specifies the input to be provided by the user, the output to be returned to the user, and the processing steps involved in producing that output

Clean Architecture

- Frameworks and drivers
  - User interface
  - Database
  - External interface
  - Web
- Interface adapters
  - Controllers
  - Gateways
  - Presenters
- Application business rules
  - Use cases
- Enterprise business rules
  - Entities

Clean Architecture visualization

- Entities are at the core
- Input and output are both in outer layers

Clean Architecture - dependency rule

- Dependence on adjacent to layer - from outer to inner
- Dependence within the same layer is allowed, but try to minimize coupling
- Anything declared in an outer layer must *not* be mentioned by the code in an inner layer
- To go from inside to outside, use dependency inversion
  - An inner layer can depend on an interface, which the outer layer implements

Identifying violations of Clean Architecture

- Look at the imports at the top of each source file

Benefits of Clean Architecture

- All the "details" (frameworks, UI, database, etc) live in the outermost layer
- The business rules can be easily tested without worrying about the outer layer details
- Any changes in the outer layers don't affect the business rules

## 10 Java

Virtual machine architecture

- **Virtual machine (VM)** - an application that pretends to be a computer
- Java has its own VM: *JVM*, that optimizes the byte code as it runs
- Each VM application is compiled for each OS so programs in the languages are *portable* across operating systems

Java conventions

- Make all instance variables `private` and create getters/setters as necessary
- `AllClassNamesAreCamelCase` with first letter capitalized
- `packages_use_lowercase_pothole`
- `variablesAreCamelCase` with first letter lowercase
- `methodsAreCamelCase` with first letter lowercase
- `FINAL_VARIABLES_ARE_ALL_CAPS` with underscores separating the words
- Always use braces, even for omething one line

Java memory model

- Primitive types store values
- Reference types store references to objects, which store values
- If a variable is the type-casted version of another, they store the reference to the same object, albeit having different classes

Types

- Java checks types at compile time, *before* the program is run

Encapsulation

- Hiding implementation details through using `private`
- With encapsulation, we can change implementation without having to change any other code

Inheritance hierarchy

- All classes form a tree called *inheritance hierarchy*, with `Object` at the root
  - Class `Object` does not have a parent; all other Hava classes have one parent
  - If a class has no parent declared, it is a child of class `Object`
  - Guarantees that every class inherits methods like `toString` and `equals`
- A parent class can have multiple child classes

Multi-part objects

- Suppose class `Child` extends class `Parent`
- An instance of `Child` has:
  1. A `Child` part, with all the data members and methods of `Child`

2. A `Parent` part, with all the data members and methods of `Parent`
3. A `Grandparent` part, ..., etc, all the way up to `Object`

- An instance of `Child` can be used anywhere that a `Parent` is legal
  - But not the other way around, since `Child` may have methods not in the `Parent` class

Shadowing and Overriding

- Suppose class `A` and its subclass `AChild` each have an instance variable `x` and an instance method `m`
- `A`'s `m` is **overridden** by `AChild`'s `m`
  - Can be used to specialize behaviour in a subclass
- `A`'s `x` is **shadowed** by `AChild`'s `x`
  - Confusing
  - Avoid instance variables with the same name in a parent and child class
- If a method must not be overridden in a descendant, declare it `final`

Casting

- The compiler doesn't run the code, it can only look at the types
- Must use casting if we want to use specific methods

Javadoc

- Placed above the method

```java
/**
 * Replace a square wheel of diagonal diag with a round wheel of diameter diam.
 * If either dimension is negative, use a wooden tire.
 * @param diag Size of the square wheel
 * @param diam Size of the round wheel
 * @throws PiException If pi is not 22/7 today
 */
 public void squareToRound(double diag, double diam) {...}
```

- Written for classes, member variables, and member methods

Direct initialization of instance variables

- Two ways of initializing instance variables:
  1. Initialize inside constructors
  2. Initialize in the same statement where they are declared (direct initialization)
- Limitations of directinitialization:
  1. Can only refer to variables that have been initialized in previous lines
  2. Can only use a single expression to compute the initial value

Creating an object

1. Allocate memory for the new object
2. Initialize the instance variables to their default values

- `0` for `ints`
- `false` for `booleans`
- etc.
- `null` for class types

3. Call the appropriate constructor in the parent class
    - Either `super(arguments)` or the no-arg constructor
4. Execute any direct initializations in the order in which they occur
5. Execute the rest of the constructor

Interfaces and classes

- Have interfaces on the left and classes on the right

```
        List<String> ls = new ArrayList<>();
```

- We can choose a different implementation of `List` at any point

Generics

- `class Foo<T>` introduces a class with type parameter `T`
- `<T extends Bar>` inroduces a type parameter that is required to be a descendant of the class `Bar` or `Bar` itself
    - This `extends` can also mean `implements`
- `<?  extends Bar>` is a type parameter that can be any class that extends `Bar`
    - We never refer to this type later, so it doesn't have to be named
- `<?  super Bar>` is a parameter that can be any ancestor of `Bar`

Naming conventions of generics

- Very short, preferable a single character
- All uppercase
- Specific suggestions:
    - Maps: `K, V`
    - Exceptions: `X`
    - Nothing particular: `T` for one, or `S, T, U, T1, T2, T3` for several

Collections

- `List`
- `Map`
- `Set`

## 11 Exceptions

Exceptions

- Report *exceptional conditions*
- These conditions deserve exceptional treatment

Exceptions in Java

- To throw an exception:

```
    throw Throwable;
```

- To catch an exception and deal with it:

```
    try {
        statements
    } catch (Throwable parameter) {
        statements
    }
```

- If we aren't going to deal with exceptions, add `throw Throwable` to method header

Benefits of using exceptions

- Less programmer time spent on handling errors
- Cleaner program structure since exceptional situations are isolated
- Seaparation of concerns: pay local attention to the algorithm being implemented and global attention to errors taht are raised

When *not* to use exceptions

- The situation that the exception reports is *not* exceptional
- Exceptions are *local*

Methods of `Throwable`

- Constructors:
  - `Throwable()`, `Throwable(String message)`
- Other useful methods:
  - `getMessage()`
  - `printStackTrace`
  - `getStackTrace`

`Error`s and `RuntimeException`s

- `Error`
  - Indicates serious problems that a responable application should not try to catch
  - Abnormal conditions that should never occur
- `RuntimeException`

  - Called *unchecked* because we don't have to handle them
  - A lot of methods throw them

- We don't have to handle either

What *not* to catch

- `Error` - we aren't expected to handle these
- `Throwable` or `Exception` - catch something more specific

What *not* to throw

- `Error` or any subclasses - these are for unrecoverable circumstances
- `Exception` - throw something more specific

Extending exception

- Can have custom exception class subclassing `Exception`
- Can also have an exception class *inside* a class that needs that exception

Classes inside other classes

- Two types of class inside another class:
  1. Static nested class - use the `static` keyword
     - Cannot access any other members of the enclosing class
  2. Inner classes do not use the static keyword
     - Can access all members of the enclosing class (even private ones)
- Nested classes increase encapsulation since we won't need to use them outside their enclosing classes

Checked vs unchecked exceptions

- Use checked exceptions for conditions from which the caller can reasonably be expected to recover
- Use run-time exceptions to indicate programming errors (e.g. precondition violations)
- If the programmer could have predicted the exception, don't make it checked

## 12 Packages

Packages in Java

- **Package** - a folder that contains related classes and packages
- Full name of a class include the package names, e.g. `java.util.ArrayList`

Packaging

- By layer
  - Clean architecture layers
  - Recommended
- By component
  - **Component** - a package that encapsulates a set of related functions or data
  - Recommended
- By feature
- Inside/outside

Benefits of packaging

- Organization
- Encapsulation (with appropriate access modifiers)

## 13 UML

UML

- Unified Modelling Language
- A way to draw information about software, including how parts of a program interact

Example

| NameOfClass |
|---|
| - privateDataMember: String[] |
| + publicDataMember: boolean |
| # protectedDataMember: long |
| + NameOfClass(p1: String[], p2: boolean, p3: long) |
| - privateMethod(param: List): int |
| + publicMethod(param: Set): void |
| # protectedMethod(param: Map): char |

Notation

- Data members:

```
      name: type
```

- Methods:

```
      methodName(param1: type1, param2: type2, ...): returnType
```

- Visibility:
  - - private
  - + public
  - # protected
  - ∼ package
- Static: underline
- Abstract method: *italic*
- Abstract class: *italic* or <<abstract>>
- Interface: <<interface>>
- Inheritance: arrow with solid line
- Interface implementing: arrow with dotted line

## 14 Design Patterns

Goals of object-oriented design

1. Low coupling
    - **Coupling** - interdependencies between objects
2. High cohesion
    - **Cohesion** - how strongly related the parts are inside a class

Design pattern

- General description of the solution to a well-established problem using an arrangement of classes and objects
- Patterns describe the shape of the code erather than the details
- Decrease coupling and increase cohesion

Iterator design pattern

- Context
    - A container/collection object
- Problem
    - Want to way to iterate over the elements of the container
    - Want to have multiple, independent iterators over the elements of the container
    - Do *not* want to expose the underlying representation: should not reveal *how* the elements are stored
- Design pattern
    - Our iterator class implements the `Iterator` interface, which has methods `hasNext()` and `next()`
    - Our iterable class implements the `Iterable` interface, which has method `iterator()` that returns our iterator class

Observer design pattern

- Problem
    - Need to maintain consistency between related objects
    - Two aspects, one dependent on the other
    - An object should be able to notify other objects without making assumptions about who these objects are
- Design pattern
    - Our observer class implements the `Observer` interface, which has method `update(o: Observable, arg: Object)`
    - Our observable class implements the `Observable` interface, which holds the observers, and have the following methods:
        * `addObserver(o: Observer)`
        * `deleteObserver(o: Observer)`
        * `deleteObservers()`

      * `hasChanged()`
      * `setChanged()`
      * `clearChanged()`
      * `notifyObservers()`
      * `notifyObservers()`

```
if hasChanged() {
    for (o : observers) {
        o.update(this, agr);
    }
}
clearChanged();
```

Strategy design pattern

- Problem
  - Multiple classes that differ only in their behaviour (e.g. using different versions of an algorithm)
  - Want the implementation of the class to be independent of a particular implementation of an algorithm
  - The algorithms could be used by other classes, in a different context
  - Want to *decouple* the implementation of the class from the implementation of the algorithms
- Design pattern
  - All of our concrete strategies implement the `Strategy` interface, which has a method for the algorithm interface
  - Our context class holds an instance of `Strategy`, and has method for the context interface (which is the algorithm interface for whichever strategy it holds)

Dependency injection design pattern

- Problem
  - We are writing a class, and we need to assign values to the instance variables, but we don't want to hard-code the types of the values
  - We want to allow subclasses as well
  - **Hard dependency**: using operator `new` inside first class to create an instance of a second class
  - Want to avoid hard dependency
- Design pattern
  - Instead of creating hard dependencies, allow instances of the second class to be passed, or *injected*, into the first class
  - This allows subclasses of the second class to be injected in as well

Simple factory design pattern

- Problem
  - One class wants to interact with many possible related objects
  - We want to obscure the creation process for these related objects
  - At a later date, we might want to change the types of the objects we are creating

- Design pattern (an example)
  - We have different shapes: `Circle`, `Rectangle`, `Square`, all of which implement `Shape`
  - `ShapeFactory` has `getShape()` method, which creates a shape
  - Our main class holds an instance of `ShapeFactory` and can ask it to create shapes

Façade design pattern

- Problem
  - A single class is responsible to multiple "actors"
  - We want to encapsulate the code that interacts with individual actors
  - We want a simplified interface to a more complex subsystem
- Solution
  - Create individual classes that each interact with only 1 actor
  - Create a façade class that has all the intended responsibilities, and references to each individual class
  - Delegate each responsibility to the individual classes

Builder design pattern

- Problem
  - Need to create a complex structure of objects in a step-by-step fashion
- Design pattern
  - Our director class has a construct method, which requires a builder
  - Our builder interface requires methods `buildPart1()`, `buildPart2()`, . . . , `buildPartX()`, `getProduct()`
  - Our concrete builder class implements the builder interface, and construct a product, as obtained by `getProduct`

# 15 Serialization and Persistent Data

Serialization

- Converting an object into a format in which it can be:
  1. Stored
  2. Transferred
  3. Reconstructed (deserialized)
- Allows for data to persist between runs of a program

Serialization in Java

- Java provides the `Serializable` interface
- Any class implementing it can be serialized
- Can save to a file (.ser)
- Each attribute of the object must implement `Serializable`

Alternatives of Java serialization

- Save to a...
  - txt file (custom format)
  - csv file (standard format)
  - json file (standard and expressive format)
  - xml file (standard and expressive format)
  - Database

## 16 Code Smells

Bloaters

- Code, methods, and classes that are so large so that they are hard to work with
- Accumulate over time as the problem evolves
- Examples
  - Long method
  - Long parameter list
  - Large class
  - Data clumps (different parts of the code contain identical groups of variables)
  - Primitive obsession (Using too much primitives instead of small objects)

Object-orientation abusers

- Incomplete or incorrect application of object-oriented programming principles
- Examples
  - Switch statements (too complex)
  - Refused bequest (subclasses uses only some of the methods or properties inherited from its parent)
  - Temporary field (which get their values only under certain circumstances, else are empty)
  - Alternative classes with different interfaces (two classes perform identical functions but have different method names)

Change preventers

- If we need to change something in one place, then we have to make many changes in other places too
- Program development becomes complicated
- Examples
  - Divergent change (having to change many unrelated methods when we make changes to a class, consequential)
  - Shotgun surgery (making any modifications requires also making small changes to many different classes, must make the changes together)
  - Parallel inheritance hierarchies (when we create a subclass for a class, we also need to create a subclass for another class)

Dispensables

- Something pointless and unneeded
- Examples
  - Comments
  - Data class
  - Duplicate code
  - Dead code
  - Lazy class (that is either very small as a consequence of refactoring or designed to support future development that never got done)

    - Speculative generality (unused class, method, field, or parameter)

Couplers

- Excessive coupling between classes
- Examples
    - Feature envy (a method accesses the data of another object more than its own data)
    - Inappropriate intimacy (class that uses the internal fields and methods of another class)
    - Message chains (a series of calls `a->b()->c()->d()`)
    - Middle man (class performing only one action, delegating work to another class)
    - Incomplete library class (library does not have the desired features)

## 17 Regular Expressions

Regular expressions

- Describes a pattern that appears in a set of strings
  - Any such string *matches* or *satisfies* the regular expression

Example: `^[a-z][a-zA-Z0-9]*`

- `^` – the pattern must start at the beginning of the string, called an *anchor*
- `[]` – choose one of the characters listed inside
- `*` – 0 or more of whatever immediately precedes it
- `$` – the end of string, another anchor

If there are no anchors, then the pattern will find the substrings that match

Special symbols

- `.` – matches any character
- `\s` – a single space
- `\t` – a tab character
- `\n` – a new line character

Inside a square bracket, `^` matches any character *except* the contents of the square brackets

- E.g. `^aeiouAEIOU` matches anything that isn't a vowel

Character classes

| **Construct** | **Description** |
|---|---|
| `.` | any character |
| `\d` | a digit `[0-9]` |
| `\D` | a non-digit `[^0-9]` |
| `\s` | a whitespace char `[\t \n \x0B \f \r]` |
| `\S` | a non-whitespace char `[^\s]` |
| `\w` | a word char `[a-zA-Z_0-9]` |
| `\W` | a non-word char `[^\w]` |

Quantifiers

- `*` – 0 or more
- `+` – 1 or more
- `?` – 0 or 1
- `{n}` – exactly $n$ copies of the pattern
- `{n,}` – $n$ or more copies of the same pattern
- `{n,m}` – $n, n+1, \ldots, m-1, m$ copies of the same pattern

Escaping a symbol

- We want symbols to show up in the string that otherwise have meanings in regular expressions

- To “escape” the meaning of the symbol, we write a backslash \in front of it
- E.g. `\.` is the period itself

Repeition of exact characters

- To repeat the *same* characters, we use **gropus** which are denoted by round brackets
- We escape the number of the group we want to repeat
- Groups are assigned the number of open brackets that precede them
- E.g. `^(([ab])c) \2 \1` repeats the inner group first, then the outer group
  - Matches `acaac` and `bcbbc`

Logical operators

- `|` means “or”
- `&&` means the intersectino of the rnage before the ampersands and the range that appears after
- E.g. `[a-t&&[r-z]]` would only include the letters `r`, `s`, and `t`

## 18 Ethics

Target demographics

1. When we imagine the average user, whom do we think of?
2. Who else could benefit from our program?
3. Who might find it difficult to use our program?

Disability

- Physical or mental impairment
- Personal or social limitation

Impairment

- Biological (based on a theory of human function)
- Statistical (based on deviation from some defined average)

Limitation

- A limit on the set of activities that a person can perform
  - Basic movements
  - Complex actions
  - Social activities

Medical model of disability

- When someone has a disability, the limitations are *caused* by the physical or mental impairment
- Examples of software design
  - Braille translation software
  - Software that runs a simplified phone interface
  - Voice recognition software

Social model of disability

- Suggests that the limitations are *not* caused by the physical or mental impairments themselves, but instead by the *way they interact with the human world*, whether through stigma or environmental
- Examples of software design
  - Automatic captioning
  - Ability to zoom in
  - Ability to unlock with fingerprint
  - Dark theme

An intermediate view

- Some diabilities are better understood as medical, while others are better understood as social

Principles of universal design

1. Equitable use

2. Flexibility in use
3. Simple and intuitive
4. Perceptible information
5. Tolerance for error
6. Low physical effort
7. Size and space for approach and use

Accommodation

- On the medical model, this will mean removint the traits that cause the limitation
    - E.g. prosthetic limbs, cognitive therapy
- On the social model, this will mean redesigning the physical and social environment to make the triat no longer result in limitations
    - E.g. wheelchair ramps, sign language, Braille

Reason for accommodation: *morality*

- *Moral particularism* – our moral judgments are irreducible to one another
    - Each circumstance is unique and there are no general rules that guide us from situation to situation
- *Moral generalism* – we can and should ground our individual moral judgments in broader principles

Moral principles

1. Utilitarianism
    - We should and always act in a way that *maximizes* the amount of *happiness* and *minimizes* the amount of *pain*
    - Every unit of happiness/pain counts equally
2. The capability approach
    - Ten "central human capabilities": life; bodily health; bodily integrity; senses, imagination, and thought; emotions; practical reason; affiliation; other species; play; and control over one's environment
    - *Everyone* must possess the bare minimum of these (ten central) capabilities necessary for a dignified life

Reason for accommodation: *law*

- Legal requirements, i.e. Charter of Rights and Freedoms
- Morality and law overlap, but they are not identical
- E.g. colour should not be used as the only visual means of conveying information

Reason for accommodation: *professionality*

- To treat all persons fairly and with respect, to not engage in harassment or discrimination, and to avoid injuring others

Clean Architecture and accessibility

- Clean Architecture makes it easier to update the accessibiity features of the program because they should mostly be located in the outer layer

# 19 Boundaries

Boundaries

- Separate software elements from one another
- Lines are drawn between things that matter aned things that don't
  - E.g. the GUI doesn't matter to the business rules, so there should be a line between them
- Motivated the layers in Clean Architecture
- Single-responsibility principle tells us where to draw our boundaries

## 20 Model View Controller

Three related patterns

- Model View Controller (MVC)
- Model View Presenter (MVP)
- Model View ViewModel (MVVM)

MVC pattern

1. The user sees the view
2. The user interacts with the View, which immediately asks the Controller to take over
3. The Controller manipulates the Model
4. The Model updates the View

MVP pattern

1. The user interacts with the View, but the View immediately asks the Presenter to take over
2. The Presenter manipulates the Model
3. The Presenter updates the View

MVVM pattern

1. The user interacts with the View
2. The View updates information to the ViewModel
3. The ViewModel represents the state of the View
    - The View is very thin, and is bound to the ViewModel
4. The ViewModel passes control to the Model
5. The Model updates the ViewModel, which is observed by the View

Comparison

- MVP: the Presenter has a reference to a View and directly instructs it what to do
    - All presentation logic is in the Presenter
- MVVM: View updates are managed through *binding* to the ViewModel, observing the changes
    - Usually cleaner, and allow UI/UX designers to focus purely on the GUI
    - Programmers manage and update the state of the ViewModel, which is directly reflected in the View
- MVC: the Controller doesn't directly update the View, the Model takes care of that

## 21 Interviewing

Steps to the interviewing process

1. Preparation
2. Application
3. First screen
4. Onsite interviews
5. Offer

Preparation

- Network
  - E.g. LinkedIn, social events
- Experience
  - E.g. side projects, open source contributions

Application

- Find companies and teams
  - E.g. via direct search, referrals, job websites
- Apply to the position
  - 1-page resume
  - Contact the hiring manager directly
- Optimize for learning, mentorship, impact, brand
  - Do not optimize for salary

First screen

- Phone screen
  - Basic questions (i.e. interests, goals, etc.)
  - Review of resume and prior experience
  - Tech questions
- Tech challenge
  - Coding challenge
  - Write clean code (i.e. conventions, tests, naming, design patterns, comments, etc.)

Onsite interviews

- Usually 1-2 onsite interviews
- Technical and behavioural
- Ask questions to understand and clarify
- Look up the team *before* the interview

- Questions they want answers to
  - What is it like to work with you?
  - Can you learn?
  - Do you take initiative?

Offer

- Understand the offer
  - Compensation
  - Team, direct supervisor
  - Vacations, benefits, etc.

Leetcode

- Repository of coding problems that emphasize data structures and algorithms
- Be aware of time and space complexity: aim for below $n \log n$

## 22 Floating Point

Floating point

- A way to represent and work with real numbers on a digital computer
- Since digital computers are finite, floating point only *approximates* the real numbers

$$\underbrace{\pm}_{\text{sign}} \overbrace{1.1234}^{\text{mantissa}} \times \underbrace{10}_{\text{base}} \wedge \overbrace{\ 8\ }^{\text{exponent}}$$

IEEE-754 floating point standard

- A *technical standard* for floating point
- 32 bits for a float, 64 bits for a double
- Specifies *formats* and *operations* for floating point arithmetic in computer systems
- *Exception conditions* are defined and handling of these conditions is specified

Purpose of the IEEE-754 floating point standard

- Computation with floating-point numbers will *yield the same result* whether the processing is done in hardware, software, or a combination of two
- The results of the comptation will be *identical*, independent of implementation, given the same input data
- Errors will be reported in a consistent manner regardless of implementation

Numeric types in Java

- `float` and `double` primitives for real numbers
- `byte`, `short`, `int`, and `long` primitives for integers
- We generally use `double` and `int`
- Also `BigDecimal` and `BigInteger` classes that are slower but provide much more precision
- The above classes are subclasses of `Number`

Algorithms for adding numbers

- Normal summation
  - $\mathcal{O}(n)$ error
  - $\mathcal{O}(n)$ running time
- Sort, then normal summation
  - $\mathcal{O}(n)$ error but lower than the previous algorithm
  - $\mathcal{O}(n \log n)$ running time
- Kahan summation algorithm
  - $\mathcal{O}(1)$ error
  - $\mathcal{O}(4n)$ running time
  - Achieved by error compensation