1 Subject Introduction

Nothing

2 Java Tour

Nothing

3 Classes and Objects 1

All programming languages has: calculation, selection/flow control, iteration, and abstractions.

Abstraction is the process of creating self contained software that allows the solution to be generalized and more general purpose.

In procedural languages, functions are the main abstractions. In OOP languages, Abstract Data Types (ADT) containing data and functions are the abstractions. A java class is an ADT.

A class is a generalization of an entity. It contains attributes and methods and defines a new data type.

Objects are instances of classes. It contains state that is dynamic. We can say that X is type C, X is an object of class C, X is an instance of class C. More specifically, an object is a specific instance of a class, while an instance is a realized class.

Data abstraction is the technique of creating new data types (classes) that are suited for an application. A java class determines the data (attributes) that the data type holds and methods (operations) that the type can use.

Encapsulation is the ability to group data (attributes) and methods that manipulate the data to a single entity.

The instance variables are class properties/attributes that are unique to each instance of the class, they represent state.

Java classes are derived data types instead of primitive data types, because they all derive from the Object class.

Variables to derived data types are always references to the underlying object/instance.

Null is the java keyword for nothing, they cannot be accessed in getting its instance variables or members.

New is the keyword to instantiate a new class. Instantiation allocates memory and creates an object of a class.

Java's automatic garbage collector will automatically free unused/unreferenced objects.

The java main static method is first ran when the program starts. It is passed with the given CLI arguments.

4 Classes and Objects 2

Accessors (Getters) and Mutators (Setters) are methods that encapsulate the instance variables. The IDE can automatically generate them

Constructors are methods used to initialize the object (setup). They have the same name as the class, cannot return any values, and can have multiple.

Without an explicit constructor, java will set the values to be the default values depending on its data type.

Method overloading is a form of polymorphism (same method, different behavior), where methods can have the same name but distinguished by their signatures

- Number of arguments
- Types of arguments
- Order of arguments

All methods can be overloaded.

Polymorphism is the ability to process objects differently depending on their data type.

This is a keyword that is a reference to the calling object, the object that is running the method.

Static variables are shared variables between all objects, one copy per class. We can access them in all instances and static methods.

Static methods are shared methods between all objects. They can only access static variables and other static methods, but can be called by everyone.

Standard methods are frequently used and can be overriden.

- The equals method is used to compare objects.
- The toString method returns a string representation of the class that is automatically used when converting to a string.
- A copy constructor creates a new instance based on an existing object.

5 Classes and Objects 2

Package allows grouping classes and interfaces into bundles, which can be handled together with an accepted naming convention.

Packages are indicated by the directory paths. Nested packages has the directory paths separated by dots in the package statement.

The classes in the same directory belongs with an unnamed default package, and thus does not need an import statement to access classes in the same package.

The class path environmental variable dictates the root package that the java jre looks to find all packages. Multiple paths means that the first one in the list will be considered first.

Information hiding is the ability to hide class implementation details from outside code.

Access control is preventing outside classes to modify attributes inside the class. We use visibility modifiers to do that.

The visibility modifiers are

	Class	Package	Subclass	Outside
public	yes	yes	yes	yes
protected	yes	yes	yes	no
default	yes	yes	no	no
private	yes	no	no	no

The convention is to set the attributes to be private and accessed through getter and setter methods which are public. Methods that nothing else should invoke are marked private.

Reasons for information hiding

- Safely seal data within the class
- Prevents relying on class implementation details
- Protect against accidental wrong usage
- Clean and elegant
- Provides a consistent interface

Mutable classes has public mutator methods that can change the instance variables, its instances/objects are mutable objects. (The attributes are not final)

Immutable classes has no methods that change its instance variables.

A class can delegate its responsibilities to other classes. It can invoke methods in other objects through containership. This is an association relationship between the classes.

A delegation through association is simply using another class as an attribute in one class. An association relation is when a class is an attribute of another class.

Wrapper classes for primitives allows primitive data types to be boxed and unboxed, and pretend that they are classes. A wrapper class in general provides extra functionality to a wrapped class.

Primitive wrappers have static methods like parseInt. Every wrapper class has a parse method that converts a string into that data type.

Boxing is the process of converting a primitive type to its equivalent wrapper class. Boxing and unboxing are automatic.

6 Bagel and Software tools

Software version control is a way to manage concurrent versions of software artefacts like documents, code, data.

It has benefits of keeping different versions of code saved, with simple reversions. Also allows committing code locally before upstream.

The git model consists of

- Repository, a collection of files to be stored by the vcs
- Remote repositories are on a remote server, local repository are maintained on the local directory
- Staging area, where changes are first staged before local commit
- Working directory, the codebase where development happens usually at the head of branches
- Branches are different versions of codebase being worked on as a branch of commits. Master is the main branch pointing to the live codebase.

The actions you can take in the git model are

- Clone/Pulling from the remote repository to the working directory
- Staging from the working directory to the staging area
- Committing from the staging area to the local repo
- Pushing from the local repo to the remote repo

Git merges incorporate changes from one branch to another. You can merge by sending a merge request (pull request) to merge a branch onto master.

Breakpoints are lines where the code will pause, to allow one to inspect the state, variable and methods.

Watch expression is an expression you want to see the value of.

Step over jumps to the next line of code, skips all method call in the lin.

Step into / Jump out, jumps into the code of the method called in the line or jump out of the method call.

Read Bagel documentations.

An opensource project is a piece of software where the original source code is made freely available and may be redistributed or modified.

Google hires opensource developers.

7 Arrays and Strings

7.1 Primitive arrays

An array is a sequence of similar type elements.

We can declare an array without initializing it. Accessing this array is a compile time error.

Arrays are pointers in that they dont need to have a length in the type, the length is only needed when initializing/creating one.

We can create arrays from initializer lists or from allocation.

Assigning arrays to another performs a shallow copy.

Multidimensional arrays are array of arrays. They can be declared and initialized just as 1D arrays. The first some dimensions may have length in the initialization, while the later dimensions can be empty and will remain being null.

Arrays being references has a default value of null.

7.2 Object arrays

We can use arrays to store objects. Note that the default values of object arrays after initializing are null.

Array methods are

- indexing using square brackets

- length using .length
- equality using .equals
- Array resizes requires creating a new array than copying the old values.
- Array sorting from Arrays.sort sorts the array in place ascending.
- Array printing from Arrays.toString.

To loop over each element in an array, we use a for each loop.

7.3 Strings

A string is a class and a data type. It is a java class made of a sequence of characters.

The double quotes character is reserved. We have to escape them to use them in strings by a backslash.

The string class is overloaded so you can add or append to a string. If either operands of + is a string, it will convert the other into a string using .toString.

Strings are immutable. Every string method returns a new string.

String methods

- Substring presence, use .contains
- Substring location, use .indexOf
- Substring, use .substring

String equal using the double equals compares references. Constant strings in java at compile time are put into the same string constant pool and thus the double equals compare works. But this does not hold if the second string is constructed not by a constant, so use the .equals instead.

8 Input and Output

8.1 Command Line Arguments

The args variable in the main function contains all command line element strings. Enclose argument with spaces with double quotes in bash.

Command line arguments are parameters passed into the java program at startup.

Command line arguments has no interactivity, and is only good for program configuration flags.

8.2 Scanner

Scanner wraps the java input streams and allows us to do IO. To create a scanner, pass the standard input stream into it as an argument. We should only create this once!

We can use the scanner to read the next typed input string (automatically parses it). We can also use it to read the next word by the .next method.

To read a line until the newline, use the .nextLine method. This is the only method that absorbs a newline (which is dropped in the return value).

Scanner does not automatically downcast types when using next methods, and will throw a runtime exception when it can't parse string into the data type (Except that it downcasts doubles to floats).

To check if parsing is possible, there is the .hasNext method that checks if there are any input, and .hasNext type methods that checks if the next input is of type. We should type check when parsing inputs.

8.3 Reading Files

The FileReader object provides a low-level interface to read a file. The BufferReader class wraps around that to provide a scanner like interface to reading the file.

A resource management try catch is used so that the closable variables are created in parentheses and automatically closed no matter any exceptions.

We can also wrap FileReader into a Scanner and use methods on a scanner. This is slower than bufferreader because the scanner class has smaller buffer sizes.

A csv file has columns separated by commas. It is like a spreadsheet.

8.4 Writing Files

We wrap the FileWriter object with the PrintWriter class over the filename. This provides a System.out style printing interface to write the file.

We do not need to write this in exams.

9 Inheritance and Polymorphism

Inheritance is a form of abstraction that permits generalization of similar attributes and methods of classes.

In inheritance, the superclass/parent/base class provides general information to its child classes; the subclass/child/derived class inherits common attributes and methods from the parent class.

Properties of inheritances

- Define common attributes and methods in the base class
- Derived class will automatically contains all public/protected instance variables and methods in the base class
- Additional methods/instance variables can be defined in the derived class
- Allows reusing code
- Derived class should be the more specific version of the base class

Java extends indicates that a class inherits from another. Inheritance defines a “IS A” relationship. Only use inheritance when it makes sense, the subclass can add additional methods and override methods in the base class.

The reference and the object can have different types if the object type is a derived class of the reference type. We only have method access of methods defined in the reference, but the actual methods called will depend on object type.

The keyword super method is used to call the parent constructor. It is also used to access parent class base methods in overrides.

Super must: only be used in a child constructor or overrides, it must be the first statement in the child constructor, and parameter types of the super method must match that of the base class constructor.

9.1 Inheriting Methods

There are some methods that are shared across all derived classes, and some methods are that are specific to each derived class.

If a method is defined only in the base class, it will always get called.

If a method is defined in both classes, the subclass has overriden the method and all function calls on the subclass object will call the subclass method instead.

Overriding is declaring a method that exists in the base class and derived class, with the same signature. Methods can only be overriden by derived class. Overloading is declaring multiple methods with the same name but different signatures.

The properties of overriding

- The derived class can extend the base class methods
- The derived class can change the behavior

- Creates an interface when we store the subclasses as parent class reference with different behaviors, calling subclass methods from a superclass reference.

Note that always, even in the base class method, the overrided method in the subclass will be called if the object is a derived class.

Overriding can't change return types, but it can return a derived class of the base method return type.

The annotation @Override is optional to mark an override method.

10 Inheritance and Polymorphism

10.1 Visibility of Inheritance

The visibility modifier of an overriden method can only be equal or a greater access than the parent method. Private methods cannot be overriden, only protect and public can be overriden.

If you don't want derived class to override a method, use the final keyword on the method.

Derived class cannot call private methods, nor access private attributes of the parent class. Derived class can access and call protected attributes and methods.

Defining attributes that are protected creates privacy leaks for the derived class may modify them in unintended ways. We should instead expose ways to access and set the parent attributes through getter and setter methods.

Shadowing is defining attributes in the derived class with the same name as the attributes in the parent class. Don't do it. The variables will be separate and the one that is used depends on the reference type to the object instead of the object type.

Variable shadowing also applies when variables with the same name are defined in different scopes. This is very confusing.

Remember that public means it is accessible everywhere, private is accessible only in the class, default is in the class and in the package, protected is in the class, in the package, in subclasses.

10.2 Object class

Every class in java inherits from the object class. All classes have a toString and equals method from the parent object class.

The default toString method prints out the class type only. We can override the toString method with our own.

The default equals method only compares the reference/memory location. We override the equals method to make it compare on the value of the objects. Note that the parameter in the equals method is type Object. To implement the equals override, we must check for null, same object type using getClass, then downcasting and comparing values.

To compare types of classes, we have

- getClass returns an object of type Class that contains details of the object class
- isinstance returns whether the object has the class type or the type of a class derived from that class

Upcasting is when an object of a derived class is casted to a base class reference. Downcasting is casting a reference to the base class to a reference to the derived class. This only makes sense when the object is actually the derived type.

Polymorphism is the ability of objects and methods to take on many forms, using them in many different ways. They include

- Overloading as Ad Hoc polymorphism
- Overriding as Subtype polymorphism
- Substitution (using derived class in a base class reference) as Subtype polymorphism
- Generics as parametric polymorphism

10.3 Abstract class

Abstract class is a class that contains common attributes and methods and represents an incomplete concept, but cannot be created. A concrete class is a class that is not abstract and can be created.

To define an abstract class, put abstract in front of the class that makes it incomplete. To mark a method abstract, we put abstract in front of it.

Each subclass must provide their own implementation of abstract methods through overriding, the base class does not have any logic in abstract methods. An abstract class may have no abstract methods, a class with any abstract methods must be declared abstract.

Forms of inheritances are

- Single inheritance
- Multiple inheritance (multiple super class)
- Hierarchical inheritance (one super class, many sub class)
- Multilevel inheritance (derived from derived)

- Hybrid inheritance (both multiple and hierachical inheritance)
- Multipath inheritance (inheritance of multiple superclasses through multiple paths)

Java does not support multiple inheritances. Java supports multiple interface inheritances.

11 Interfaces and Polymorphism

Interfaces are abstract concepts that can't be instantiated. It has the features

- Contains only constants and abstract methods
- Defines behavior and actions
- Classes can implement interfaces
- Represents a "can do" relationship, classes implement the interface if it can do all the actions defined

For an interface in java

- Methods never have any code
- All methods are implied abstract and public
- All attributes are implied static final public

A class can implement an interface using the "implements" keyword. Concrete classes that implement an interface must implement all methods it defines. Classes that don't implement all methods must be defined abstract.

Interfaces can have a default implementation of a method, this can be overriden or not in class implementations. The "default" keyword creates a standard implementation of a method that does not need to be implemented if the standard implementation is valid.

If a class implements multiple interface with the same method signature/default implementation, it is a compile time error.

Interfaces can be extended, which forms a "is a" relationship between interfaces, this is used to make more specialized interfaces. Interface can extend multiple other interfaces.

Classes can only inherit one class, but can implement multiple interfaces.

Subtype polymorphism applies to interfaces too. So we can upcast a class object to the interface it implements.

11.1 Sorting

We use `Arrays.sort(Object[] array)` to sort. It uses the fact that a class implements the `Comparable<ClassName>` interface. This interface requires that the class

- Can be compared with objects of the same class (or subclass) through the `compareTo` method.
- Allows the class objects to be sorted

The `compareTo` method defines an ordering of the class objects. It compares exactly two objects of the same type and returns a negative integer if `this` is less than the other object, zero if they are equal, and a positive integer if `this` is greater than the other.

If we `Arrays.sort()` on array of objects with no Comparable implementations, we through a compiled time error.

11.2 Inheritance and Interfaces

In general between inheritance and interfaces

- We use inheritance if the object is a super class, that the derived class has a common parent and are related
- We use interface if the object share similar functionality, but are not related.

We use inheritance if one object is another, we use interfaces if one object can do the same things as another.

Use inheritance when we are passing similar methods and attributes to subclasses, use interfaces when we are grouping objects with similar actions.

12 Classes and Relationship

Learning to design a software.

First step before writing code is to design the system, and then implement the system following the design.

The algorithm to design a software is

- Identify classes (Extracting nouns)
- Identify class relationship
- Refine class relationship
- Develop a class diagram (combining classes and relationship)

- Represent using an accepted notation (UML)

UML is the unified modeling language. It is a graphical modeling language that can represent object oriented analysis, design, and implementation.

Note that UML modeling does not imply java classes and methods, they are abstract ideas that are language independent.

12.1 Classes

UML models a class by the class name, attributes, and operations (methods). A class is drawn as a bordered box with optional compartments

- The class name is a string
- The attributes has name, $+/-$ for public private, types, default value, multiplicity (size of attribute)
- The methods has name, $+/-$ for public private, and function signature in parameters and return types

Static attributes/operations are underlined and placed last in the list.

Abstract classes are written with the name in italics.

The privacy/visibility modifiers are $+$ for public, $\sim$ for package, # for protected, and $-$ for private.

The multiplicity denotes the expected ranged of elements in the collection. It has both a lower bound and an upper bound (represented by one number if the bounds are equal). We can use a star on the upperbound to denote any arbitrary integer above the lower bound. For example, a multiplicity n denotes a bound $n..n$, while a multiplicity of $a..b$ denotes the closed interval from a to b. The bounds must be non-negative integers.

If the multiplicity is along an element with text, the range is placed in square brackets after everything; if the multiplicity is used for class relations, they are written without the brackets.

Note that we only represent attributes/variables with a primitive type, attributes/variables that are derived types uses relationship lines instead.

12.2 Interfaces

Interfaces are drawn similar to classes, but with `<<interface>>` in the first line above the name.

12.3 Relationships

To represent class relationship in UML, we use four common types of relationship

- Association
- Generalization (inheritance)
- Realization (interface)
- Dependency

12.4 Association

Association is a has-a (containment) relationship between objects, it allows objects to delegate tasks to other object. It is shown by a solid line between classes with the arrow pointing to the object we have as an attribute. We do not write the attribute!

Association arises either by containment, or logical combination. Note that association does not require either side to be attributes of another, the two classes maybe in the same common map instead.

The properties of association has

- Roles on each side of association line along with multiplicities
- Name of the association written in the middle
- Unidirectional with single arrow, and bidirectional without any arrows (or with two arrows)
- Self association which is an association with itself
- Multiple association with other classes with multiple lines

A class can be related to another in the following ways

- one to one
- one to many
- one to bounded interval
- one to exactly n.
- one to infinity

We will represent these by $a..b$ implying a multiplicity between $[a, b]$. a star is used for any integer.

Navigability is the arrow placement of an association relationship, and represents whether the instance can be reasonably accessed by the other end at runtime. A navigable end is represented by an arrow, an un-navigable end is represented by a cross, and nothing implies unspecified navigability (doesnt matter).

Navigability should not be related to ownership, but it kinda implies it. We can add a visibility modifier to the navigable end (before role) to show the visibility of the attribute.

Binary association links exactly two classes together.

12.5 Aggregation

Association with aggregation implies that there is an association, but the objects lives independently. It implies that one class is related to another, but does not own it. The hollow diamond is located on the aggregate side (the side where when removed, the other side remains). Aggregation can be viewed as a simple namespace/groupings of other classes and the diamond is located on the grouping.

Aggregation is a binary relation, and is asymmetric for only one end can be an aggregation. The aggregation path is transitive and forms a DAG.

12.6 Composition

Association with composition implies an association where the objects (parts) cannot exist without its dependency (whole) existing, and is deleted without the whole. The full diamond is located on the whole end.

Composition is a binary relation, implies a whole/part relationship with the part at most associated with one whole class, and deleted when the whole is deleted. Often, there can only be one whole and multiple parts.

Confusingly, we can have a multiplicity 0..1 on the whole/composite side of the composition, implying that the part may exist without the whole, but is deleted/relieved when the whole is removed.

12.7 Generalization/Inheritance

A generalization relationship suggests that instances of the specific class are also instances of the general class, an “is a” relationship. Generalization is owned by the specific class. It is represented by a hollow arrow on a solid line. The arrow points to the general class.

Generalization represents a single relationship, while inheritance is the set of generalization relationship/structure for a given class. A class is allowed to have multiple generalizations/inheritance (although restricted when we consider only java).

12.8 Dependency

Dependency is a directed weak relationship between classes, it shows that a class requires,needs or depends on other elements for specification or implementation, and changing one class may impact the other class. It represents a supplier-client relationship, where the supplier provides functionality to the client with the client being incomplete/dependent on the supplier.

Dependencies are represented by a dashed line with line arrow head pointing from the client to the supplier.

Types of dependencies include

- Usage, where a class uses an element by creation/instantiation/calling for its full definition and implementation
- Abstraction/Realization, which states that the two elements represents the same concept but at different levels of abstraction

The class factory depends on the class itself, with the arrow pointed at the class. The class realizes the interface with the hollow arrow pointed at the interface.

Notice that the arrow head is lined if it is a usage dependency, and hollow if it an abstraction/realization dependency.

12.9 Realization/Interfaces

A realization relationship is represented by a dashed line with a hollow arrow. The arrow points to the interface.

We can create classes and relationship from a specification (creating UML from spec).

13 Generics

Java allows classes, interfaces, and method definitions to have parameter types. They are called generics.

Generics allow code reuse and enable generic logic to be written for all classes.

The comparable interface uses generics.

In generic classes, T is a type parameter/variable. When we use the class, every instance of T are replaced with actual types (intuitively, not literally).

Non-generics requires us to check for the method data types in runtime every time, while generics checks for data types in compile time.

13.1 ArrayList

ArrayList is a generic class in the collections package that overcomes the limitation of arrays.

Arrays have limits

- Finite length
- Resizing is manual
- Requires effort to add and remove elements

When instantiating array lists, we don't have to specify the types in the RHS if we've done so on the LHS. The LHS type can also be a derived parametric type of the RHS type (list of child cast to list of parent).

ArrayList can do

- Iterated in a for each loop
- Automatic resizing
- Can insert, remove, get, and modify elements at every index
- Overrides toString
- Cant be indexed using square brackets

Limitations of arraylist

- Does not shrink automatically, and can consume more memory than required; we have to manually `trimToSize()` to release the excess memory
- Cannot store primitive data types

13.2 Defining generic classes

A generic class is a class defined with an arbitrary/generic type for fields, parameters, or return types.

The type parameter is included in angular brackets after the class name in the heading. We often use a single uppercase letter, but we can use any non-keyword identifier (any possible variable names).

The unbound type parameter can have any reference type. A generic class is used just like any other class.

We use bounded type parameters (bounded generics) to guarantee a class' behavior. We state that T extends classes or interfaces in the generic declaration (split by & symbols for multiple bounds).

A generic method accepts arguments,or returns object with an arbitrary type. A generic method can be defined in any class, the type parameter is local to the method. We write the parametric type in angular brackets before the return type.

Pitfalls of generics

- We can't instantiate parameterized objects (cant new)
- We can't create arrays of parameterized objects (cant new arrays)

14 Collections and Maps

Java has two abstract data structure frameworks. Collections is a framework that contains ordered data types, and maps is a framework that contains key-value pairs. All these abstract data types allows storing, accessing, and manipulate objects in them.

14.1 Collections

The main data structure for collections is the array list.

We can downcast an array list of a derived class to an array list of base classes. It is allowed to add any derived class and iterate through the list and use methods defined on the base class (allows polymorphism and overriding).

We can sort an array list if the objects inside implements the comparable interface. Then we can call `Collections.sort()` to sort in place the array list (it automatically uses the `compareTo` methods).

For a more flexible sorting method, we use the Comparator interface with signature `Comparator<T>` with a method `int compare(T a, T b)`. We can create a new class that implements the comparator interface for the array list object, then instantiate the that realized comparator class as the second parameter to `Collections.sort()`. This allows us to have multiple comparator classes to sort the array list of objects depending on the attribute we are sorting for.

Because we only need the comparator classes when sorting, it is too verbose to name these comparators and put them in separate files. We can use anonymous inner classes to solve this: it is a class created on the fly (in a method) without a new file or class name to which only a single object is created.

These anonymous inner class can implement interfaces, and the syntax to create one is `new Interface<T>() { ...implementation... }`. We only use anonymous inner classes if we are using it once in the method, and it allows us to create temporary classes that implement some interfaces to pass into methods.

The collection hierarchy has the main interfaces of: Collection, Set, List, Queue. The main classes are: TreeSet, LinkedList, ArrayList, Priority Queue.

The collection framework has common methods of

- .size to get the collection size
- .contains to check for containment
- .add to add element
- .remove to remove elements
- .iterator to get an iterator
- for each loop on the collection
- .get to retrieve indexed collections, only for arraylists and friends

The most useful elements in the collections framework are

- ArrayList are better arrays
- HashSet to ensure elements are unique
- Priority queue is an automatic sorted queue
- TreeSet is an ordered hashset that allows fast lookup of unique elements in order

14.2 Maps

The maps hierarchy contains the interface Map, and the classes: HashMap, LinkedHashMap, TreeMap, HashTable, Properties.

HashMap stores the keys in an unordered manner, TreeMap stores the keys in a sorted way.

The common operations are

- .size
- .containKey to check if the key exists
- .containValue to check if the value exists
- .put to add/update key value pairs

- .remove to remove a key
- .keySet returns a set of all keys
- .entrySet returns a set of all key value pairs (a single key value pair is a Map.Entry)
- .get to retrieve the value for a key

All maps have two generic type parameters K, V which are the key types and value types.

Without generics, we either implement lists/maps with all as objects or rewrite the algorithm for every new data type. Generics gives us the flexibility to reuse code for any types, and it allows objects to keep their type so compilers can detect the errors in compile time (preventing run time errors).

15 Design Patterns 1

Three simple steps to be a good software designer

- Learn the fundamentals of: programming language, algorithms and data structures
- Learn design paradigms and principals: structured design, object oriented design
- Study and mimic the designs of experienced designers: reuse solutions that are already solved

Design pattern is the systematic documentation and analysis of re-occurring design solutions to design problems.

The recurring nature of problems makes the solution useful to software developers, and publishing the solutions in the form of patterns allows the solution to be reused without reinventing the wheels.

The work on design patterns and their documentation are done by the "Gang of four".

15.1 Analysis and publishing a pattern

The attributes and specification of a design pattern involves

- Intent, why the pattern exists
- Motivation, a scenario that highlights the need for the pattern
- Applicability, situations where you can use the pattern
- Structure, a graphical representation of the pattern
- Participants, list of classes and their roles in the pattern

- Collaboration, how the objects in the classes interact
- Consequences, a description of the results, side effects, and tradeoffs in the pattern
- Implementation, example of solving a problem
- Known uses, specific, real world uses of the problem

We are not tested on remembering all of the specifications for all design patterns in the exam.

15.2 Singleton

A singleton pattern ensures that a class only has one instance and provides a global point of access to it.

The implementation has a private constructor, a private static instance attribute, and a public static `getInstance` method that creates and returns the one instance.

Analysis of singleton

- Intent to ensure that the class has only one instance and is accessible everywhere
- Motivation is cases where we want only one instances and has easy access to the object
- Applicability, use when a single instance of a class is required
- Participants, just the singleton class
- Consequences, use with caution because it could result in bad design when used inappropriately
- Known uses are CacheManager class, PrinterSpooler class

15.3 Factory Method Pattern

Factory method solves the problem of creating objects without tight coupling, and creating customized versions of the class with different object creations. It defines a separate operation/factory method to create an object, and creates the object by calling the method which has implementations in the subclasses.

The keywords are

- Factory is a general method to create objects
- Product is the abstract class that generalizes the objects being created/produced by the factory

- Creator is the abstract class that generalizes the objects that consumes/produces the products. It generally has some operation that invokes the factory method and produces the product.

The factory method pattern has the creator class provide an abstract factory method to create an abstract product. We inherit the creator class by implementing the factory method to create a subclass/specific product.

Problems that the factory method pattern solves

- Code duplication
- Requires inaccessible information
- Not abstracted
- Class don't care about figuring out which object to instantiate
- Classes are rigid and fragile

Its advantages are

- Delegates object creation to subclasses
- Abstracts the object creation using a factory method and abstract product
- Encapsulates classes by allowing subclasses to determine the product they need (separating the different versions into different classes implementing different factory methods)

Analysis of factory method

- Intent to generalize object creation
- Motivation, loading player objects when a game loads
- Applicability, When sibling classes depend on and creates similar objects
- Collaboration, concrete creator objects invokes the factory method in order to produce their desired produces
- Consequences, object creation in base class is decoupled from the specific object required

16 Design Pattern 2

Template method and strategy patterns allows us to build generic algorithms that can be extended, adapted, and reused. They solve the problem of separating a generic algorithm from its details.

The two techniques that are used to develop generic components are refactoring (strategy) and generalizing (template method), which is identifying recurring code and replace with generic code. The two main OO design technique that achieves this are inheritance and delegation.

Template methods use inheritance to separate the details, and strategy pattern uses delegation for its detail implementations.

16.1 Template methods

Define the generic algorithm and expose abstract implementation methods in the base class. Any specific implementation of the algorithm is done in the subclass that implements the abstract methods and evokes the generic algorithm.

The analysis is

- Intent to define a family of algorithms, encapsulate each one
- Motivation is to build generic components that are easy to extend
- Applicability, where the algorithm has invariant parts in the base class and allows subclass to implement the varied parts
- Consequence, all algorithms in the same family must have the same interface

16.2 Strategy pattern

The problem with template methods is a strong dependency to the base class, for the implementations in the subclass cannot be reused in other similar implementations.

The base class has a generic algorithm using the details provided by another class object (the strategy), which is a concrete class that has an interface/abstract class with the implementation. To define a specific implementation of the algorithm we implement the interface and pass the concrete class to the algorithm base class as an argument.

The algorithm class is the context class, the implementation is the strategy class.

The benefit of the strategy pattern over template methods is that multiple algorithms using similar implementations/strategies can reuse the same interface, and concrete strategies for one algorithm can be reused for another algorithm. This is not possible in template methods because of the tight coupling between the implementation derived class.

16.3 Observer pattern

For situations where many objects (observers) depends on the state of one object (subject).

For that subject, this one-to-many dependency is inflexible because it tightly couples the subject with particular observers.

Observer pattern decouples the subject and observers using a publish-subscribe style communication pattern. (Subject publish changes to observers, and observers subscribe and filters the changes).

The subject is an important object whose state determines the actions of other classes. The observer monitors the subject in order to respond to its state.

The subject has methods to register, unregister, and notify the observers with a list of observers. The observer is an interface/abstract class that has a notify abstract method, and each relevant observers must implement this interface and the notify method. The subject publish state changes by calling the abstract notify method on each observer.

The benefits of observer patterns are

- Automatically notifies observers
- Decouples the subject from the observer class
- Powers most event driven program
- Clear responsibilities, where subject focuses on publishing state and not worry about the observers except that they exist
- Observers can be dynamically added and removed from the subject with zero effect to the subject (a publish subscribe model)

Java has the observer interface with an `void update(Observable obj, Object state)` method. The observable abstract class has the methods `void addObserver(Observer obj)` to subscribe an observer and `void setChanged()` and `void notifyObservers()` to publish the change to the subscribers. Any observer would implement the observer interface, and any subject would inherit from the observable abstract class.

16.4 Types of Design Patterns

They relate to commonly solved classes of patterns

- Creational patterns deal with solutions in object creation — Singleton, Factory methods
- Structural patterns deals with the structure of classes and their relationship — Adapter, Bridge
- Behavioral patterns deals with interaction among classes — Strategy, Template method, Observer

17 Exceptions

Errors are mistakes made when writing the code. Errors while developing and typing the program are categorized as

- Syntax errors, where the code isnt legal, detected by the editor/compiler
- Semantic errors, code runs, but results in incorrect output, identified through software testing
- Compilation errors, error in the compilation
- Runtime errors, an error that causes the program is end prematurely, identified through execution

Types of runtime errors

- Dividing a number by zero
- Accessing element out of bounds
- Store incompatible data elements
- Converting a string to another type
- File errors, opening a file that doesn't exist, opening a file without the right permissions

Ways to prevent runtime exceptions

- Do nothing lmao
- Explicitly guard against dangerous and invalid conditions: defensive programming. The prolem is that we need to explicitly protect against every possible error conditions, some conditions don't have a backup paths, not nice to read, poor abstraction and bloated code
- Use exceptions to catch error, recover from them or gracefully end the program

Exceptions are error states created by a runtime error in the code. Exceptions are also objects created by java to represent the error that occurred. Exception handling contains code that actively protects the program in the case of exceptions.

To handle an exception, we use a try catch finally block. If there is an exception in the try block, the catch blocks may catch the exception, and the finally block is ALWAYS ran no matter which path is taken AFTER the path is taken:

- Try attempts to execute code that may result in exceptions
- Catch deals with a specific exception, could be recovery or failure

- Finally performs clean up no matter which condition?

We can chain exception catches to deal with different exception. The catch matching is from top to down, matching using the base class (instance of) method. This implies that we should put the most specific exception first and broader exceptions later. Java won't compile if you put a specific exception after the broad exception.

Usually, we wont need to worry about the order of different exceptions that are not subclasses of each other.

To generate an exception we use throw to respond to an error state by creating and throwing an exception object, either using an existing exception class or a custom one. Throws indicates that a method has the potential to create an exception, and can't handle it or the exact response varies for different applications.

If a method contains an exception throw that is not a RuntimeException class, we MUST mark the method with throws. If it throws RuntimeException, we dont need to mark the method with throws.

Unchecked exceptions can be safely ignored by the programmer (most java exceptions are unchecked, and you arent forced to protect against it). Checked exceptions must be explicitly handled by the programmer, and the compiler will error if the checked exception is ignored. The tree of exceptions/throwables are

- Throwable (Checked) for the root
- Exception (Checked), containing RuntimeException (unchecked) and other exceptions
- Error (unchecked), containing other unchecked errors

To define a custom exception we must extend the exception(error) class. All exception class will have two constructors, one empty and one with a message string, but we can create more.

All checked exceptions are handled by catch or declare: enclosing close with a try-catch block, or declaring that a method may create an exception. Both techniques can be used in the same method for different exceptions.

We should only use exceptions when a method encounters an unusual or unexpected case that cannot be handled easily in the method.

A try with block creates closeable variables in parenthesis after the try keyword, and calls close on the variables in the finally block automatically.

18 Software design and testing

18.1 Coding styles

General code standards includes

- Use consistent layout like indentation and white space
- Avoid long lines
- Beware of tabs
- Lay out comments and code neatly
- Sensible naming of variables, methods, and classes
- Avoid duplicating code
- Use comments to explain code sections

Comment styles and conventions

- Intended primarily for yourself, and other developers
- Code should be self-documenting, readable without extra doc
- Should tell the story of the code
- Sufficient to piece together the algorithm even if the code is removed
- Attached to blocks of code, corresponding to the steps in the algorithm
- Comment before code, they are prologues that introduces the code

Javadoc a cli tool that is in the jdk which automatically generates doc. Javadoc comment has the properties

- Starts with multiline comments
- Compiled to HTML
- Document packages, classes, methods and attributes
- Use tags to declare parameters and return values
- Intended primarily for developers
- Should document how to use and interact with your classes and methods
- Not the same as commenting code

18.2 Design Principals

Design principals in the OOP context includes

- Encapsulation, information hiding, delegation, inheritance, realization, polymorphism
- Modeling classes and relationships
- Design patterns

These principals are applications of general software design principals (that doesn't require OOP) applied in a OOP context.

The following are the set of general software design principals.

Modularity is the decomposing the problem to units (modules) that are easy to understand, manage and re-use on their own. In OOP, modules are represented as classes. We combine classes using relationships to solve bigger problems.

Cohesion states that modules must be designed to solve a clear and focused problem. Designs should have high cohesion. This implies that class methods and attributes are related and work together towards a common objective.

Coupling is the degree of interaction between modules. Coupling should be reduced as much as possible; designs should have low coupling. This means: avoid unnecessary associations because it increases coupling, try to pass objects as parameters (dependency vs association) to reduce coupling, interfaces promote low coupling.

Open-Closed principle states that modules should be open to extension, but closed to modification. This means that if we need to change or add functionality of the class, we should not modify the original class but rather extend it. Uses delegation, inheritance and interfaces.

Abstraction is creating abstract data types to represent problem components using classes, represents data and actions. Encapsulation states that details of the class should be hidden and private, user's ability to access the details is restricted using information hiding. Polymorphism is the ability to use an object or method in different ways. Delegation keeps the class focused by passing work to other classes, stating that computation is done in the class with the most relevant information.

Symptoms of poor design

- Rigidity, hard to modify the system because changes in one class and method cascade to others
- Fragility, changing one part of the system causes unrelated parts to break
- Immobility, cannot decompose the system into reusable modules

- Viscosity, writing hacks when solving a problem to preserves a design
- Complexity, clever code that isnt necessary right now, premature optimization
- Repetition, duplicated code
- Opacity, convoluted logic, design is hard to follow

18.3 Software Testing

To solve a bug: write print statements, debuggers, googling.

Bugs can be discovered through software testing. The phases of software testing are

- Unit testing that tests units independently before integration
- Integration and system testing, integrating units to form the system and testing the system
- Acceptance testing, validating if system support the functionality as per the requirements

Java has a structured method for unit testing.

- Units are small, well defined components of a software system with one or a small amount of responsibilities.
- Unit tests verifies the operation of unit by testing a single use case (input/output) intending for it to fail or pass.
- Unit testing is identifying bugs in software by subjecting every unit in a system to a suite of unit tests.

Unit tests should test the unit with both valid and invalid inputs, intending it to pass and fail successfully. If a unit/method is too big, we should break down the code into smaller units.

Types of unit tests are

- Manual testing tests manually in an ad-hoc manner. This is difficult to reach all edge cases and not scalable for large projects
- Automated testing tests using custom built automatic testing software that is less human dependent and faster and more reliable.

JUnit automatic testing is a piece of software that is used to do unit testing. Its purpose is to provide a standardized way to write and execute tests, allowing developers to easily ensure their code behaves as expected. In the context of JUnit,

- Assert is a true or false statement that indicates the success or failures of a test case.
- TestClass class is a class dedicated to testing a single unit. The naming convention is to use the form ClassTest where Class is the unit name

The JUnit asserts should be statically imported; the JUnit tags (decorators) are imported as classes.

The JUnit assertion methods are

- assertArrayEquals checks for character array equality
- assertTrue
- assertFalse
- assertEquals checks if the two parameters are equal
- assertNotEquals

The format of the `assertEquals` is often `assertEquals(T expected, T actual, String? message)`. For doubles, there is an extra third argument for a delta.

JUnit tags are decorators on methods in TestClasses

- `@Test` marks a method to be a test method, indicating that it should be executed as a test case
- `@BeforeEach` marks a method to be executed before every test method in the class, used to set up common objects before the tests
- `@AfterEach` marks a method to be executed after each test method, used to clean up resources or teardown after each test
- `@BeforeAll` marks a method to be executed once before all test methods, used for one-time setup operations
- `@AfterAll` marks a method to be executed once after all test methods

For lifecycle tags/decorators, we can store the setup variables as instance variables.

All test methods should by convention start with test. Inside each test, it should call assertions.

Automated test and test suites are useful to test your program. The unit tests would need to be created accurately, however, and it is the developer's responsibility in doing so.

Advantages of JUnit. Large teams and open source devs should always use automated testing because they are

- Easy to set up

- Scalable
- Repeatable
- Not human intensive
- Powerful
- Find bugs

19 Event Driven Programming / Javascript

Sequential programming is a program that is run from top to bottom, starting at the beginning of the main method and concluding at the end.

This is useful for static programs which does not depend on other inputs, with constant, unchangeable logic, and the execution is the same each time.

Event driven programming uses events and callbacks to control the flow of a program's execution based on changes to the program states. We've seen this in observers, GUI, and exception handling.

It is a programming style that uses the signal-response approach. It is often provided by a framework: JavaFX and web frameworks. This style is asynchronous, where the programmer will deal with events that are generated at unknown times.

State are the set of properties of an object or device. Events are created when the state of an object or device is altered. Callback/listener is a method/function that is triggered by an event.

A component with state may have many listeners. Each listener may respond to different type of events, and multiple listeners can observe the same event. Listeners must register with the event generator component in advance.

Sending an event is called firing the event. An event handler is a method in the listener that specifies what happens after receiving the events it observes.

In sequential programming, the execution order is sequential, the orders are only changed by program logic through looping, conditionals, or method invocation.

In event driven programming

- Objects are created that can fire events, the listener objects are created to react to events
- The method that is ran next depends on the next event

- The program itself does not determine the order in which things happen; the events determine the order of the program
- Handlers are never explicitly called by the developer, rather invoked automatically when an event the handler observes is fired

19.1 JavaFX

In GUI, the events we may respond to are: mouse, keyboard, touch, controller. We can implement a lot of features if we can respond to those events.

JavaFX quickstart

- Main application extends Application
- Call launch to start the application
- Start method provided a stage, stage is a window object that holds a main scene
- A scene contains a root node, a node is either a stackpane or an element with a tree structure.

The event handler is done by adding `.setOnAction(listener)`. The listener interface contains a callback method with an action parameter that listens/handles to all actions on the node.

This approach is better because

- Encapsulates classes by hiding their behavior and expose a subject for other classes to subscribe to
- Avoid explicitly sending information on state change, instead passes these actions to the hanlder
- Easily add and remove behaviors to subjects classes, easily add and remove behaviors to listening classes

Real world examples: GUI, javascript, and embedded systems and hardware interrupts. They are usually accompanied/supported by a software development framework.

An object-oriented application development framework is a set of classes that represent reusable designs. The framework consists of abstract classes and interfaces. We can then implement and extend these classes to meet the customized application needs. Example frameworks: java IO, java collection/map framework, java fx.

The key features of OOP frameworks

- Extensibility, has extendable abstract classes. Changable aspects are hook methods

- Inversion of Control. Traditional library has the application control the execution flow and acts as the master. With application development framework, the framework acts as the master
- Frameworks uses design patterns as building blocks

20 Advanced Java and OOP

20.1 Enumerated Types

Enum is a class consisting of a finite list of constants. It can be used to represent a fixed set of values, and must list all values inside the enum definition. Outside of the list of constants, they are classes with methods and attributes. The methods and attributes are declared after the list of constants separated by a semicolon.

The values of an enum are accessed statically because they are static constants. Their values are enum objects just like any other object.

Enums can have constructors and attributes, implying that the list of constants must be supplied with an argument when declaring them inside the enum, which can be set in the constructor to an attribute.

Enums comes pre-built with

- Default constructor
- toString
- compareTo. The default sorting/comparing function on enums are dictionary order on the enum string.
- ordinal, which is the order of the constant as declared in the enum

They are classes, so we can add and override any methods or attributes we want. We cannot extend in an enum nor extend from an enum.

20.2 Variadic arguments

Variadic methods are methods that can have a variable (unknown) number of parameters. They are not the same as overloaded methods where multiple methods each deal with a fixed number of parameters, but rather have a single method to deal with any number of parameters. Variadic methods implicitly convert the variable length input parameters into a variadic array.

We declare a variadic parameter by placing three dots after the type name before the parameter name; variadic parameters must be the last parameter in the method. The variadic parameter is treated (but not actually) exactly like a java array with that type inside the method.

20.3 Functional Interfaces

Functional Interfaces is an interface with a single abstract method, also called a Single Abstract Method interface. It is declared as an interface annotated with the `@FunctionalInterface` decorator, so that if the functional interface has more than one non-static method it raises a compiler error. The annotation is however optional just like `@Override`.

The purpose of functional interfaces is enabling lambda expressions and method references. It is a concise way to implement single method interfaces, making the code more readable and maintainable. Some functional interfaces are: Comparator, Callable, Runnable.

Functional interfaces are also valuable on their own for they define a clear contract with a single abstract method, making them ideal for scenarios where we need to pass behavior as a parameter

- Callback mechanism: defines a single action to perform
- Strategy patterns: implement strategies that can be swapped
- Stream API: uses functional interfaces like Predicate, Function, Consumer to manipulate lists

Functional interfaces are valuable even without lambdas because

- They provide a clear contract
- Ensure consistent behavior definition
- Simplify implementations using anonymous inner classes
- Maintain compatibility with frameworks and API
- Facilitate testing and documentation
- Support design patterns like Strategy and Command

The `Predicate<T>` functional interface represents a predicate, a function that accepts one argument and returns true or false. It has a `boolean test(T t)` method that takes a single object and returns a boolean. It can be combined with other predicates using the `and`, `or`, and `negate` methods (which takes another predicate as input and returns a new predicate).

The `UnaryOperator<T>` functional interface represents a unary (single argument) function that accepts and returns a value with the same type. It has the `T apply(T t)` method that takes and returns the same data type.

20.4 Lambda functions

A lambda expression is a technique that treats code as data (treat code as an object). It allows us to instantiate an interface with an object without implementing it. The syntax is a single arrow lambda which is then assigned to a functional interface.

A lambda expression takes zero or more argument (source variables) and applies an operation to them. The parenthesis around the source variables and braces around the operation are optional.

Lambda expressions can often be used in place of Anonymous classes, but they are not the same thing. Lambda expressions can only be used on functional interfaces because they are instances of functional interfaces, allowing us to treat the functionality of the interface as an object.

20.5 Method References

A method reference is an object that stores a method on a class. It can take the place of a lambda expression if the lambda expression is only used to call a single method on the value. A method reference can be stored like a lambda expression to a variable of a functional interface type.

The different types of method references are

- Static methods with the syntax `Class::staticMethod`
- Instance methods with `Class::instanceMethod` or `object::instanceMethod`. If it is from the class, the first argument is the class instance; if it is from the object, the this argument is populated.
- Constructors with `Class::new`

Instead of calling the method, a method reference stores a function with implied arguments that are given when the method reference is called.

20.6 Streams

A stream is a series of elements in sequence that are automatically put through a pipeline of operations. We can apply a series of operations on a stream by chaining stream operations.

Streams are techniques that allow us to apply sequential operations to a collection of data. The operations are

- map that converts input to output
- filter that selects elements by a predicate

- limit that performs a maximum number of iteration (limits the stream size)
- collect gathers all elements and output in a list, array, or string
- reduce aggregates a stream into a single value