Class

1. Object contains attributes which store its state and methods which implement its behaviour.
2. Class contain attributes and methods to describe behaviour of an object.
3. Instances are objects created from a class.
4. An instance/field/member variable identifies state of an object's instance.
5. Constructor is invoked when a new object is created. Every class has a constructor.

Static

1. A static/class variable's value is shared by all instances of a class.
2. Static methods are methods that exist outside class and can only access static variables.
3. Non-static methods can access static variables.

Polymorphism/Inheritance/Composition

1. Polymorphism means an operation may be performed in different ways in different classes. Requires multiple methods of same name. Occurs when a parent class reference is used to refer to a child class object.
2. Inheritance/"is-a" relationship is one object acquires properties of another object.
3. Superclass/parent class contain features common to a set of subclasses.
4. Subclass/child class inherits from superclass but also contain unique features.
5. Composition/aggregation/"has-a" relationship is class contain references of another class.

Abstract/Interface

1. Abstract class cannot be instantiated, has no implementation and contains abstract methods which are method declarations without body.
2. Class that inherit from AC must override all methods from AC.
3. Interface is a collection of abstract methods and cannot be instantiated.
4. Can extend from only one abstract class but can implement from multiple interfaces.

Overriding/Overloading

1. Function overriding is subclass provide its own implementation of a method that is already provided by its superclass.
2. Function overloading is a class has multiple functions with same name but with different method signature.

Modularity/Encapsulation

1. Modularity means code can be constructed entirely of classes.
2. Encapsulation means details can be hidden in classes. This gives rise to information hiding where programmers don't need to know all details of a class.
3. Encapsulation also prevents uncontrolled access.

Access control

1. Public – any class can access.
2. Protected – only code in package or subclasses can access.
3. (blank) – only code in package can access.
4. Private – only code written in class can access.

Error

1. Compile-time error/syntax error is an executable version of program is not created.

2. Run-time error is a problem occur during program execution that causes program to terminate abnormally.
3. Logical error is a program may run but produce incorrect results.

Java

1. Within instance method or constructor, *this* is a reference to current object – the object whose method or constructor is being called.
2. A variable declared as final is used to declare a constant that always holds the same value and cannot be changed.
3. A method declared as final cannot be overridden.
4. A class declared as final cannot be inherited and its methods cannot be overridden.
5. Scanner scan = new Scanner(System.in); input = scan.nextLine();
6. @Override keyword asks complier to check whether there is a method in superclass to be overridden.
7. super keyword is used to purposely invoke a superclass's method.
8. Package is a namespace that organise classes and interfaces by functionality.

Conversion/Casting

1. Assignment conversion is value of a data type is assigned to another data type.
2. Data promotion/conversion happens automatically when operators in expression convert their operands. E.g. result = sum / count.
3. Casting, e.g. result = (float) sum / count.
4. Upcasting is substituting a subclass instance for its superclass. Safe because a subclass instance has all properties of its superclass.
5. Downcasting is substituting a superclass instance for its subclass. Not safe. Requires casting operator to inform compiler the class type to downcast to (else compile-time error will occurs).

Types of software projects

1. Corrective projects fix defects.
2. Adaptive projects change system in response to changes in OS, DB etc.
3. Enhancement projects add new features.
4. Perfective projects change system to make it more maintainable (functionality remain same).

Activities in software projects

1. Domain analysis consists of gathering requirements, analysing requirements and preparing requirements specification which details instructions about how the software should behave.
2. Design decides how to implement the requirements such as systems engineering (what is in hardware and what in software), software architecture (divide system into subsystem and how subsystems interact), detailed design of internals of a subsystem, UI and DB design.
3. Modelling creates representations of the domain or software using UML diagrams.

Types of requirements

1. Requirement is an aspect of what the system must do or a constraint on the system's development.
2. Functional requirements describe what the system should do.
3. Quality requirements are constraints on the design to meet specified levels of quality.
4. Platform requirements are constraints on environment/technology of the system.
5. Process requirements are constraints on project plan and development methods.

Object types

1. Entity object represents persistent information tracked by the system.
2. Boundary object represents interaction between user and system.
3. Control object coordinates boundary and entity objects. Collect information from boundary objects and dispatching to entity objects.

OO design principles

1. Single-responsibility principle (SRP) – a class should have only one responsibility.
2. Open-closed principle (OCP) – a class should be easily extendable without modifying the class itself. Implemented using abstract classes and then create concrete classes to implement their behaviour.
3. Liskov substitution principle (LSP) – every subclass should be substitutable for their superclass.
4. Interface segregation principle (ISP) – when we write our interfaces we should add only methods that should be there.
5. Dependency inversion principle (DIP) – write class details based on abstractions instead of vice versa.
6. Divide and conquer – divide up the system so that you can better handle its complexity.
7. High abstraction allows you to understand the essence of a subsystem without having to know unnecessary details.

Increase cohesion

1. Cohesion is indication of relationships within a module.
2. A module has high cohesion if it keeps together things that are related to each other and keeps out other things that are irrelevant.
3. Functional cohesion (best) – keep together codes that computes a particular result and keep out everything else.
4. Layer cohesion – keep together facilities for providing/accessing a set of related services.
5. Application programming interface (API) – set of procedures through which a layer provides its services.
6. Communicational cohesion – keep together modules that access/manipulate same data.
7. Sequential cohesion – keep together procedures in which one procedure provides input to the next.
8. Procedural cohesion – keep together different activities which are executed sequentially.
9. Temporal cohesion – keep together activities that are related in time.
10. Coincidental cohesion (worst) – keep together activities that have no meaningful relationship to each other.
11. Best – functional.
12. Satisfactory – communicational and sequential.
13. Reserved for application specific – logical, temporal, and procedural.
14. Worst – coincidental.

Reduce coupling

1. Coupling is indication of relationships between modules.
2. Coupling occurs when there are interdependencies between one module and another.
3. When interdependencies exist, changes in one place will require changes somewhere else.
4. Content coupling (Worst) – occurs when one component is free to modify data that is internal to another component.
5. Common coupling – occurs when multiple components have access to same global variable.
6. Control coupling – occurs when one procedure calls another using a ‘flag’/’command’ that explicitly controls what the second procedure does.
7. Stamp coupling – occurs when one class is declared as the type of a method argument.

8. Data coupling – occurs when the types of method arguments are either primitive or simple library classes. The more arguments a method has, the higher the coupling.
9. Routine call coupling – occurs when a method calls another method.
10. Type use coupling – occurs when a module uses a data type defined in another module.
11. Inclusion/import coupling – occurs when one component imports a package (in Java) or includes another (in C++).
12. External coupling (Least bad) – occurs when a module has a dependency on OS, shared libraries or hardware.

Reasons to use UML

1. Divide a complex system into discrete pieces to be understood easily.
2. Since UML is not platform specific, a complex system can be understood by multiple developers working on different platforms.

UML notation

1. Things – entities to be modelled, whether concrete or abstract.
2. Relationships – connections between things.
3. Diagrams – graphical depictions of things and their relationships.

Types of UML diagrams

1. Structural diagrams – focus on static aspects of system. Class, object/instance, component, deployment.
2. Behavioural diagrams – focus on dynamic aspects of system. Use-case, interaction (sequence/collaboration), state, activity.

Sequence vs Collaboration

1. Why use sequence diagram?
- Make explicit the time ordering of the interaction.
- Make it easy to add details to messages.
2. Why use collaboration diagram?
- Might be preferred when deriving an interaction diagram from a class diagram.
- Useful for validating class diagrams.

Reasons to use DP

1. Developing software from scratch is hard.
2. Developing reusable software is even harder.
3. Patterns and frameworks are proven solutions.

Types of DP

1. Structural patterns provide a manner to define relationship between objects or classes. Adapter, bridge, composite, façade.
2. Behaviour patterns provide a manner to define communication between objects or classes. Iterator, observer.
3. Creational patterns provide ways to instantiate single objects or groups of related objects. Builder, prototype, singleton.

Structural

1. Composite provides a common interface to treat individual objects and composition of objects uniformly.
2. Adapter converts an interface of legacy class into an interface expected by client so that legacy class and client class work together without changes.
3. Bridge separates an implementation from its interface so that implementation can be substituted, possibly during run time.
4. Façade provides a unified interface to a set of related objects in a subsystem.

Behaviour

1. Iterator provides a uniform interface to traverse a collection of object sequentially without needing to know its underlying representation.
2. Observer provides a method to represent one-to-many relationship between objects such that dependent objects are notified when an object is modified.

Creational

1. Singleton ensures a class only has one instance and provide global access point to it.
2. Builder is used to create complex object from individual parts in certain order/algorithm.
3. Prototype provides a method to clone an object without affecting performance.