Android and Software Design Fundamentals

Introduction to Android and Kotlin

- **Kotlin and Android Synergy:** Android SDK is primarily Java-based, but Kotlin code is seamlessly merged during compilation into DEX (Dalvik Executable) code, which Android devices use.
- **Kotlin's Advantages:**
  - Named after an island near St. Petersburg, Russia.
  - Similar to Apple's Swift, making iOS development learning easier.
  - Succinct, leading to less code and fewer errors.
  - Considered a first-class Android language by Google.
  - Used in popular apps like Kindle, Evernote, Twitter, Expedia, Pinterest, and Netflix.

- **Android's Foundation:** Android runs on a specially adapted Linux operating system, providing a user-friendly interface while hiding underlying complexities.
- **Android API (Application Programming Interface):** This acts as an interface, making it easier for developers to access hardware and software features in a programmer-friendly way, analogous to a car's accelerator pedal.
- **Object-Oriented Programming (OOP):** Kotlin is object-oriented, using concepts like classes (blueprints) and objects (instances) to structure code based on real-world things for reusability.
- **Android Studio:** An Integrated Development Environment (IDE) that simplifies code compilation and integration with the Android API.
- **Android Resources:** Images, sound, UI layouts, and text content are stored in separate files (often XML for UI and text) to facilitate changes and multi-language support.
- **Packages and Classes:** Packages group related code (like folders), and classes are blueprints contained within packages. Functions within classes perform specific actions.

Software Design Principles and Concepts

- **Design Methodology:** The process of creating a software design, involving decisions on formality, complexity, and structure. Refactoring is used to simplify or optimize designs.
- **Design Principles:** Guidelines for decomposing system functionality into modules.
- **Modularity (Separation of Concerns):** The principle of keeping unrelated aspects of a system separate so each can be studied in isolation. Systems are decomposed by identifying and encapsulating unrelated concerns into modules.
- **Module Independence - Coupling and Cohesion:**
  - **Coupling:**
    Measures the interdependence between modules.
    - **Uncoupled:** No interconnections.
    - **Loosely Coupled:** Weak interconnections.
    - **Tightly Coupled (Highly Coupled):** Significant dependence.
    - **Types of Coupling (from least to most desirable):**
      - Content Coupling: One module modifies another.
      - Common Coupling: Modules share global data.
      - Control Coupling: One module controls another's behavior via flags or parameters.
      - Stamp Coupling: Complex data structures are passed.
      - Data Coupling: Simple data values are passed.
      - Message Coupling (implied by loose coupling): Modules interact via message passing.

- **Cohesion:**
  Measures the dependence within a module's internal elements.
  - **Coincidental Cohesion (Worst):** Unrelated elements grouped together.
  - **Logical Cohesion:** Elements related by logic structure.
  - **Procedural Cohesion:** Elements related by execution order.
  - **Temporal Cohesion:** Elements related by timing.
  - **Communicational Cohesion:** Elements operate on the same data.
  - **Sequential Cohesion:** Output of one element is input to another.
  - **Functional Cohesion (Best):** Every element contributes to a single, well-defined function.

- **Interfaces:** Define the services a software unit provides and how other units can access them. An interface specification includes functionality, preconditions, postconditions, and protocols.
- **Information Hiding:** Encapsulating design decisions within a software unit, exposing only its interface to hide complexity and facilitate maintenance.

Software Design Methods and Quality Attributes

- **Popular Design Methods:**
  - **Functional Decomposition:** Partitioning functions into modules.
  - **Feature-Oriented Design:** Assigning features to modules.
  - **Data-Oriented Decomposition:** Focusing on data partitioning.
  - **Process-Oriented Decomposition:** Partitioning into concurrent processes.
  - **Event-Oriented Decomposition:** Assigning responsibility for events to modules.
  - **Object-Oriented Design (OOD):** Viewing the system as a collection of objects (entities, classes, messages, abstraction, encapsulation, inheritance, polymorphism).

- **Achieving Quality Attributes:**
  - **Modifiability:** Ease of making changes (essential due to life-cycle costs).
  - **Performance:** Constraints on system speed and capacity (response time, throughput, load).
  - **Security:** Immunity to attacks and resilience (ability to recover).
  - **Reliability:** Correctly performing functions under assumed conditions, with fault prevention and recovery.
  - **Usability:** Ease of operation, often involving user interface design in its own unit.

- **Software Requirements Specification (SRS):** Documents functional and non-functional requirements, which should be clear, actionable, measurable, and traceable.
- **High-Level Design:** Breaks down the architectural design into sub-systems and modules, showing their interactions.
- **Detailed Design:** Implements the high-level design, defining the logical structure and interfaces of each module.
- **Characteristics of Good Design:**
  - **Modularity:** Splitting software into small, single-purpose components.
  - **Maintainability:** Ease of fixing bugs or adding features.
  - **Performance:** Ability to pinpoint performance bottlenecks.
  - **Portability:** Ease of moving software to different environments.

- **Usability:** Providing a clear overview for customers.
- **Trackability:** Linking design to requirements.
- **Deployment:** Documenting where deliverables should be placed.

Software Design Levels and Architecture

- **Interface/High-Level Design:** Specifies the interaction between a system and its environment, treating the system as a black box.
- **Architectural Design:**
  Specifies major components, their responsibilities, interfaces, and relationships. It defines the overall structure without internal details of components.
  - Consists of: components, integration conditions, and semantic models.

- **Conceptual Design:** Tells the customer what the system will do.
- **Technical Design:** Informs system builders about the hardware and software needed.

User Interface Design

- **Elements of UI Design:**
  - Metaphors: Recognizable concepts.
  - Mental Model: Organization and representation of data/functions.
  - Navigation Rules: How to move between elements.
  - Look: System appearance.
  - Feel: User experience.

Introduction to Mobile Programming

- **Mobile Application Development:** Creating software for smartphones, tablets, etc., for Android and iOS. Apps can be preinstalled, downloaded, or accessed via a web browser.
- **Languages:** Java, Swift, C#, HTML5.
- **Key Features:** Uniqueness, User-Friendliness, Richness, Power, Marketing, Flawless Coding.
- **Growth:** Mobile app development is rapidly growing as organizations meet user demand for real-time access and transactions.
- **Major Platforms:**
  - **iOS:** Developed by Apple, runs exclusively on Apple products. Requires macOS for development.
  - **Android:** Backed by Google, dominant market share (around 80%), but fragmented across devices and OS versions.

- **Android vs. iOS:**
  - **Platform:** Android (Java, open-source, Linux kernel) vs. iOS (Objective-C/Swift, proprietary, Apple hardware only).
  - **Market Share:** Android has a significantly larger OS market share globally.
  - **Development:** Android generally requires more time but has a shorter review process; iOS requires less time but has a longer review process.
  - **Cost:** Android development is often less expensive.
  - **Demographics:** Android appeals more to lower-income groups, iOS to higher-income groups.
  - **Design:** Android caters to multiple device operators; iOS to single device operators.

- **Types of Mobile Apps:**
  - **Native Application Development:** Uses platform-specific SDKs and languages (Java/Kotlin for Android, Swift/Objective-C for iOS). High performance and user experience.
  - **Web Application:** Uses web technologies (HTML, JavaScript, CSS), runs in a mobile browser, OS-independent. Accessed via URL, not distributed through app stores.
  - **Hybrid Application:** Combines web technologies within a native wrapper. Can access device features, distributed through app stores.
  - **Cross-Platform Application Development:** Develops an app once for multiple platforms (iOS, Android). Saves time and effort, cost-effective, but may have performance trade-offs compared to native apps.

- **Mobile App Development Life Cycle:** Planning & Research -> Assessment -> Wireframe & Prototype -> Design -> Develop -> Test -> Deploy -> Launch -> Enhance.

Android Activity Lifecycle

- **Activities:** Fundamental components representing a single screen in an app. The Android system manages their lifecycle through callback methods.
- **Lifecycle Importance:** Preserves user data, prevents crashes, avoids memory leaks, and handles interruptions (calls, screen rotation).
- **Activity States and Callbacks:**
  - **onCreate():** Activity is created. Initialize UI and variables.
  - **onStart():** Activity becomes visible.
  - **onResume():** Activity gains focus and is interactive.
  - **onPause():** Activity loses focus (partially visible). Stop non-essential operations.
  - **onStop():** Activity is no longer visible. Release resources.
  - **onDestroy():** Activity is being destroyed. Perform final cleanup.
  - **onRestart():** Called when activity is re-started from a stopped state.

- **Saving State:** Use onSaveInstanceState() to save UI state in a Bundle for configuration changes (like screen rotation) or when the app is killed in the background. The Bundle is passed to onCreate() and onRestoreInstanceState().

Application ID and Debugging Tools

- **Application ID:**
  Uniquely identifies an app.
  - **Package Name:** Reverse domain name structure (e.g., com.example.myapp).
  - **Application ID:** Similar to package name, can differ for debug/release builds. Defined in build.gradle.
  - **Purpose:** Used for Google Play Store publishing, interacting with APIs (Firebase, Google Maps).

- **Debugging Tools in Android Studio:**
  - **Logcat:** Displays real-time logs (errors, warnings, info). Use Log.d(), Log.e(), etc.
  - **Breakpoints:** Pause code execution at specific lines to inspect variables.
  - **Layout Inspector:** Analyze UI layout in real-time.
  - **Memory Profiler:** Monitor memory usage, detect leaks.
  - **CPU Profiler:** Analyze app performance and identify CPU-intensive methods.

- **Building, Running, and Debugging Workflow:**
  1. Open project in Android Studio.
  2. Sync Gradle files.
  3. Select a device (emulator or physical).
  4. Click Run.
  5. Use Logcat and breakpoints to identify and fix issues.
  6. Rebuild and re-run.

Application Development and Emerging Technologies

- **Application Development:** The process of gathering requirements, designing, coding, testing, and improving software.
- **Emerging Technology:** New or developing technologies expected to have significant social or economic impact (e.g., AI, IoT, Blockchain, AR/VR, Cloud Computing, DevOps, Big Data).
- **System Development Life Cycle (SDLC):** A framework defining tasks at each stage of software development (Planning & Requirement Analysis, Defining Requirements, Designing Architecture, Building/Developing, Testing, Deployment & Maintenance).
- **Role of Data:** Data is a critical asset driving science, technology, and the economy in the age of Big Data.
- **Industrial Revolutions:**
  - **IR 1.0:** Mechanization, steam power, factory system (late 1700s-early 1800s).
  - **IR 2.0:** Mass production, interchangeable parts, electricity, telephone (late 1800s-early 1900s).
  - **IR 3.0 (Digital Revolution):** Digital electronics, computers, internet (late 1900s).
  - **IR 4.0 (Cyber-Physical Systems):** Robotics, IoT, AI, autonomous systems (present).

- **Types of Industries:** Primary (raw materials), Secondary (manufacturing), Tertiary (services), Quaternary (R&D).
- **Enabling Devices:** Memory devices, microprocessors, logic devices.
- **Human-Machine Interaction (HMI) / Human-Computer Interaction (HCI):** The study and design of how humans interact with machines and computers. Natural user interfaces (gestures) are increasingly important.
- **Future Trends:** 5G Networks, AI, Autonomous Devices, Blockchain, Augmented Analytics, Digital Twins, Edge Computing, Immersive Experiences.
- **SDLC Phases:** Planning, Defining Requirements, Designing Architecture, Building, Testing, Deployment & Maintenance.
- **Role of System Analyst:** Guides the project, understands requirements, prioritizes, and suggests solutions.
- **Mobile Application:** Software designed for mobile devices.
- **Key Mobile App Technologies:** Native Apps, Web Apps, Hybrid Apps.
- **Software:** Instructions and documentation that tell hardware what to do. Classes include System Software, Application Software, and Programming Software.
- **Laws of Software Evolution:** Continuing Change, Increasing Complexity, Conservation of Familiarity, Feedback Systems, Continuing Growth, Reducing Quality, Self-Regulation, Organizational Stability.
- **Software Process Models:**
  - **Waterfall Model:** Sequential phases (Requirements, Design, Implementation, Verification, Maintenance). High risk for new systems.
  - **Prototyping:** Iterative, specification and program stay in step.
  - **Spiral Model:** Risk-driven, combines Waterfall and iterative approaches. Phases include

Planning, Risk Analysis, Engineering, Evaluation.

- **Project Management:** Essential for managing cost, time, scope, and quality in software development.

Capturing Requirements

- **Requirements:** An expression of desired behavior (what the customer wants, not how). Focus on the problem, not the solution.
- **Process:** Elicitation (collecting user needs), Analysis (understanding behavior), Specification (documenting behavior), Validation (checking against user needs).
- **Stakeholders:** Users, domain experts, clients, customers, market researchers, lawyers, auditors, software engineers.
- **Types of Requirements:**
  - **Functional Requirements:** Describe required behavior (reactions to inputs, state changes).
  - **Non-Functional Requirements (Quality Requirements):** Describe quality characteristics (response time, reliability, maintainability).
  - **Design Constraints:** Pre-made design decisions that restrict solutions.
  - **Process Constraints:** Restrictions on techniques or resources.

- **Resolving Conflicts:** Prioritize requirements into Essential, Desirable, and Optional categories.
- **Characteristics of Requirements:** Correct, Consistent, Complete, Traceable, Relevant, Testable.
- **Notational Models:** Various ways to represent requirements (Natural Language, Structured Language, DDL, Graphical Notations, Mathematical Specifications, Programming Language, Decision Tables/Trees).
- **Prototyping:** Useful when customers are uncertain; demonstrates requirements visually.
- **Validation:** Checking that requirements definition accurately reflects stakeholder needs.
- **Verification:** Checking that the specification document matches the definition document.

Designing the System Architecture

- **Software Design:** Transforming user requirements into a form suitable for programming.
- **Design Process:** Conceptualizing requirements into implementation, establishing a plan to identify optimal solutions.
- **Architectural Styles:** Generic solutions for decomposing problems into software units and defining their interactions.
- **Design Patterns:** Generic solutions for lower-level design decisions.
- **Design Process Model:** An iterative process involving understanding requirements, proposing solutions, testing feasibility, and documenting.
- **Popular Design Methods:** Functional Decomposition, Feature-Oriented Design, Data-Oriented Decomposition, Process-Oriented Decomposition, Event-Oriented Decomposition, Object-Oriented Design.
- **Quality Attributes:** Factors like Portability, Usability, Reusability, Correctness, Maintainability, Reliability, Efficiency. Architectural styles aim to promote these.
- **Tactics:** Fine-grained decisions to improve specific quality goals.
- **Modifiability:** Crucial due to high post-development life-cycle costs.
- **Performance:** Response time, throughput, load capacity.
- **Security:** Immunity and resilience against attacks.
- **Reliability:** Fault-free operation, fault prevention, and recovery.
- **Usability:** Ease of operation, often involving dedicated UI units.
- **SRS:** Documenting functional and non-functional requirements.

- **High-Level Design:** Less abstract view of sub-systems and modules.
- **Detailed Design:** Implementation details of modules and interfaces.
- **Characteristics of Good Design:** Modularity, Maintainability, Performance, Portability, Usability, Trackability, Deployment considerations.
- **Software Design Levels:** Interface Design, Architectural Design, Conceptual Design, Technical Design.
- **Designing User Interface:** Involves metaphors, mental models, navigation rules, look, and feel.

Designing Modules

- **Design Methodology:** Transforming requirements into an architectural design, with potential for refactoring.
- **Design Principles:** Guidelines for decomposing functionality into modules.
- **Modularity (Separation of Concerns):** Breaking down systems into unrelated concerns, encapsulating each in a module.
- **Coupling:** Inter-module dependence (aim for low coupling).
- **Cohesion:** Intra-module dependence (aim for high cohesion).
- **Function-Oriented Design:** Uses DFDs, Data Dictionaries, Structured Charts.
- **Object-Oriented Design (OOD):** Focuses on objects, classes, messages, abstraction, encapsulation, inheritance, and polymorphism.