Design-principer

SOLID

SOLID är en minnesregel som har samlat de fem viktigaste principerna inom OO-programmering, vars syfte är att göra mjukvarudesign mer lättbegriplig, flexibel och enkel att underhålla.

Single Responsibility Principle (SRP)

A class should have at most one reason to change.

Vad fyller principen för syfte? (X)
Förändringar hålls väl avgränsade, så om funktionalitet behöver ändras minskas risken för sammanblandning (**robust**). Detta gör även klassen lättare att testa separat från annan funktionalitet (**testing**), samt att all kod som rör ett visst ansvarsområde finns på ett enda ställe (**easy access**).

Varför är det bra att följa den? (X)
Det är enklare att förstå, testa och felsöka en klass som har ett väl avgränsat ansvarsområde. Dessutom kan man ändra i klassen utan att oroa sig för att det påverkar andra delar.

Ge något konkret exempel på hur man bryter mot, eller följer principen.
Bryter: En klass *BankAccount* som hanterar uttag, insättning och saldoändring pga aktuell sparränta. Här finns tre anledningar till förändring.
Följer: Vi delar upp klassen ovan i *AccountWithdraw* , *AccountDeposit* och *AccountInterest* .

Open-Closed Principle (OCP)

Software modules should be open for extension, but closed for modification.

Vad fyller principen för syfte? (X)
Att skriva kod där ny funktionalitet enkelt kan läggas till (open for extension) samtidigt som så mycket som möjligt av existerande kod kan lämnas orörd (closed for modification).

Varför är det bra att följa den? (X)
Koden blir maintainable och extendable.

Ge något konkret exempel på hur man bryter mot, eller följer principen.
Bryter: Det finns ett switch-statement i koden som måste ändras varje gång man lägger till eller ändrar ett menyalternativ i programmet. Den är inte stängd för modifikation.
Följer: Man skriver robust kod från start med intentionen att ingen ska behöva gå in och ändra den i efterhand.

Ett bra exempel på OCP i vardagen är en smartphone. Den är *open for extension* då du kan lägga till appar för att utöka funktionalitet, men du får ingen källkod eller möjlighet att ändra telefonens grundfunktionalitet, den är *closed for modification* .

Liskov Substitution Principle (LSP) **(X)**

Beskriver när arv bör och inte bör användas.

Vad fyller principen för syfte? (X)

Den förtydligar hur en arvstruktur ska se ut (IS-A förhållande), samt sätter upp riktlinjer för när det är lämpligt att faktiskt använda sig av arv. Exempelvis ska subklasser lägga till extra beteenden / beräkningar, de ska inte ändra befintliga beteenden / beräkningar från superklassen.

Varför är det bra att följa den? (X)

Det blir en tydlig struktur vilket gör koden mer lättbegriplig samtidigt som man slipper oväntade beteenden som annars kan uppstå.

Ge något konkret exempel på hur man bryter mot, eller följer principen.

Bryter: *Human* är en subklass till *Dog* (bryter mot IS-A), eller att *Square* är en subklass till *Reqtangle.* När vi använder metoden setWidth i *Square* kommer både width och height ändras (för att det är en fyrkant), här får vi ett beteende vi inte förväntar oss då det bryter mot superklassens beteende.
Följer: *Car* ärver av *Vehicle* . Det är ett IS-A-förhållande, vi behöver inte använda metoder som bryter mot *Vehicle* 's beteende, och vi kan utöka beteendet med metoder som är specifika för *Car* .

Interface Segregation Principle (ISP)

No client should be forced to depend on methods it does not use.

Vad fyller principen för syfte? (X)

Genom att bryta ut större (fat) interfaces i flera mindre, behöver klienter inte implementera dummy-metoder som de inte ens använder. Fungerar som SRP, fast för interfaces.

Varför är det bra att följa den? (X)

Man slipper onödig kod och man slipper även skriva om koden när en metod, som man inte ens använder, byter signatur (!).

Ge något konkret exempel på hur man bryter mot, eller följer principen.

Bryter: Ha ett fat interface som innehåller flera olika beteenden. T.ex. ett interface *GenerateReport* som innehåller två metoder, generateExcel och generatePdf. Klienter som endast vill generera exceldokument tvingas ändå lägga till metoden generatePdf.
Följer: Ha många små interfaces som endast hanterar sitt eget beteende. I exemplet ovan kan man bryta ut interfacet *GenerateReport* till två olika interfaces.

Dependency Inversion Principle (DIP)

Depend on abstraction, not on concrete implementations.

Vad fyller principen för syfte? (X)

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.
Vi vill skapa så **få** och **lösa** beroenden som möjligt, samt att dessa beroenden är **avsiktliga**.
Genom att använda supertyper istället för subtyper kan vi minska beroendet av en specifik klass.

Varför är det bra att följa den? (X)

Vi får kod med låg coupling och mer flexibilitet.

Ge något konkret exempel på hur man bryter mot, eller följer principen.

Bryter: Man skapar en superklass som har starka beroenden med sina subklasser. Det orsakar stelhet och koden blir jobbig att ändra.
Följer: Subklasser skapas för att passa superklassen, och därför behöver superklassen inte ändras när vi lägger till eller ändrar i subklasserna.

Generella principer

Command-Query Separation

En metod ska antingen ha sidoeffekter (en mutator) eller returnera ett värde (en accessor), inte båda.
Asking a question should not change the answer.

Vad fyller principen för syfte? (X)

Man ska kunna anropa en query-metod (metod med returtyp) utan att oroa sig för att programmets tillstånd ändras. Här är det viktigt att ha koll på sina sidoeffekter.

Varför är det bra att följa den? (X)

Man slipper bli överraskad av oväntade beteenden.

Ge något konkret exempel på hur man bryter mot, eller följer principen.

Bryter: Metoden returnEvenNumbers tar emot en array, modifierar samma array och returnerar den.
Följer: Metoden returnEvenNumbers tar emot en array, skapar en kopia av arrayen (Mutate-by-copy), modifierar kopian och returnerar den.

Composition Over Inheritance

Favor composition over inheritance.

Vad fyller principen för syfte? (X)

Den ger lösare beroenden och mer robust kod jämfört med arv (där subklassen helt kan sluta fungera om det sker en förändring i superklassen). Koden blir även mer flexibel då Java inte stödjer multipla implementationsarv, och det är även möjligt att återanvända kod där arv inte är lämpligt.

Varför är det bra att följa den? (X)

Det som står ovan.

Ge något konkret exempel på hur man bryter mot, eller följer principen.

Bryter: Koden fokuserar på arv, inte delegering.
Följer: Koden fokuserar på delegering, inte arv.

High Cohesion, Low Coupling

Cohesion: Mått på den inre sammanhållningen i en modul (metod, klass...).
Coupling: Mått på hur starkt beroendet är mellan två moduler.

Vad fyller principen för syfte? (X)

Cohesion: Vi eftersträvar high cohesion, dvs när samtliga komponenter i modulen samverkar för att lösa sitt ansvarsområde utan att behöva andra moduler. Det ger varje modul ett väl avgränsat ansvarsområde.
Coupling: Vi eftersträvar low coupling, dvs när moduler är så oberoende av varandra som möjligt. Detta möjliggör en flexibel och modulär design.

Varför är det bra att följa den? (X)

High cohesion och low coupling lägger grunden för återanvändning av komponenter.

Ge något konkret exempel på hur man bryter mot, eller följer principen.

Bryter: Klass 1 och klass 2 håller objekt av varandra, samt att metod A i klass 1 och metod B i klass 2 refererar till den andra klassen.
Följer: Metoderna A och B beror endast på komponenter i den egna klassen för att lösa sin uppgift, och endast en av klasserna får hålla ett objekt av den andra (fast bara om det är nödvändigt).

Law of Demeter (LoD) / Principle of Least Knowledge

Don't talk to strangers.

Vad fyller principen för syfte? (X)

Vi undviker att introducera beroenden genom att: en klass bara ska känna till sina närmsta "vänner" (ofrånkomliga beroenden), och klassen ska inte heller anropa metoder hos andra än sina vänner.

Varför är det bra att följa den? (X)

LoD åstadkommer kod med låg coupling, är testbar och som kan återanvändas.

Ge något konkret exempel på hur man bryter mot, eller följer principen.

Bryter: Vi introducerar metoder som agerar på externa objekt.
T.ex. objectA.getObjectB().getObjectC().doSomething();
Följer: Vi introducerar metoder som agerar på interna objekt, istället för generiska getters.
T.ex. objectA.doSomething();

Separation of Concern (SoC)

"The SoC, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of." – Edsger W. Dijkstra

Vad fyller principen för syfte? (X)

Att separera ett program i distinkta sektioner så att varje sektion hanterar sitt eget ansvarsområde / logik. Detta kan göras på hög och låg nivå. På hög nivå kan vi bryta ut ett program i sektioner som UI, företagslogik, användarlogik och databasen. MVC är ett bra exempel på SoC där man delar upp programmet i en Modell, View och Controller. På låg nivå fungerar SoC som SRP, fast SoC hanterar mer än bara klassen.

Varför är det bra att följa den? (X)

Då kan flera programmerare jobba på olika sektioner isolerat från varandra, koden blir reuseable, enklare att underhålla, testa, med mera.

Ge något konkret exempel på hur man bryter mot, eller följer principen.

Bryter: Vi har MVC-logik utan att de är uppdelade var för sig. Vi kan även ha en metod som hanterar inloggning för företagskunder OCH privatkunder.
Följer: Vi använder och delar upp MVC som det är tänkt. Vi har även två separata metoder för inloggning, en för företagskunder och en för privatkunder.

Design patterns

Vad är ett design pattern?
Det är en generell lösning på en vanligt återkommande situation inom (mjukvaru-)design. Det är ingen färdig kod, det är en abstrakt mall för hur ett problem kan lösas. Kan ses som "formaliserade" best-practices.

Varför används det? (X)
I en given situation och kontext kan vi instansiera ett design pattern med specifika klasser, metoder, etc och få en färdig lösning. Dessutom talar man om "On the shoulders of giants." då de vanligaste design patterns är så etablerade att andra utvecklare kommer kunna förstå din kod.

Principer vs Design patterns
Principer är de mål vi vill uppnå, design patterns är våra verktyg för att nå dem.
På en glidande skala från **abstrakt** (extensibility, reuseability, maintainability) till **konkret** (färdig kod) är principer abstrakta medan design patterns befinner sig någonstans däremellan.

Creational patterns

Berör hur objekt skapas.

Factory

Creates objects without exposing the instantiation logic to the client and refers to the newly created object through a common interface.
En Factory är, per analogi, en klass vars gränssnitt består av Factory Methods.

Abstract Factory (kommer ej på muntan)

Create an abstract type for a Factory that specifies one or more abstract Factory Methods. Create different concrete Factories as subtypes to the abstract Factory type, each of which can return objects of different concrete types for each specified Factory Method.

Factory Method

Definiera ett interface för att skapa objekt, men låt subklasserna bestämma vilken klass de vill instansiera. Factory Method låter klasserna skjuta upp sin instansiering till subklasserna.

När kan man använda det?
När man vill dölja intern implementation (vilka konstruktorer och specifika klasser som används) eller låta subklasserna skjuta upp instansiering av klasser.

Hur kan man använda det?
Det som står under Factory Method.

Vilka designproblem löser man genom att använda detta pattern? (X)
Att klasser instansieras även när det inte behövs.

Singleton

Restrict instantiation of a class to a single object. Whenever an instance of the class is requested, return that one object.

När kan man använda det?

När programmet behöver en, och endast en, instans av ett objekt.

Hur kan man använda det?

1. Skapa en *private static* instansvariabel som håller instansen av klassen.
2. Skapa en *private* konstruktor så att klassen inte kan bli instansierad utifrån.
3. Ha en *public static* -metod som skapar en instans av klassen om det inte redan finns en, annars returneras en referens till objektet.

Vilka designproblem löser man genom att använda detta pattern? (X)

När det finns flera instanser för delade resurser. Att ha en instans för t.ex. en databas-anslutning, filhanterare eller skrivare förhindrar konflikter när flera anropar samma instans samtidigt.

Behavioral patterns

Berör hur objekt kommunicerar (metoder och signaturer).

Chain of responsibility

Chain the receiving objects of a request and pass the request along until a suitable handler is found.

När kan man använda det?

När man endast vill ge **ett** objekt åt gången möjlighet att svara på anropet.

Hur kan man använda det?

Genom att skapa en enda "entry point" där metod-anropet görs (helst via interface) ser det från anroparen ut som att den pratar med ett enda objekt, trots att anropet internt kan skickas mellan flera objekt innan det hanteras.

Vilka designproblem löser man genom att använda detta pattern? (X)

Att flera objekt hanterar samma anrop, när anropet endast behöver hanteras en gång.

Iterator

Ge klienter tillgång till elementen i ett sammansatt objekt som en sekvens, utan att visa hur objektet är uppbyggt internt.

När kan man använda det?

När man har behov av det som står under Iterator.
Ta som exempel hur TV-kanaler fungerar: förr växlade man mellan kanaler manuellt (1, 2, 3…) och TV:n tog ingen hänsyn till om kanalerna hade täckning. Man var då tvungen att veta vilka kanaler man behövde. Nu för tiden när man byter kanal med "Next" växlar den automatiskt till nästa kanal som har täckning. Man behöver inte längre kunna vilket nummer kanalen har (den interna representationen).

Hur kan man använda det?

I Java finns interfacen Iterator<T> och Iterable<T> som är de rekommenderade verktygen för Iterator Pattern.

Vilka designproblem löser man genom att använda detta pattern? (X)

När man vill dölja intern representation för klienterna samt ge tillgång till objekt som en sekvens.

Observer

When an object needs to notify other objects of events, without directly depending on them:
Broadcast the events on an open channel, to which any interested object may register.
"Tell, don't ask."

När kan man använda det?

Det som står under Observer.

Hur kan man använda det?

Man skapar ett observer-interface som skickar ut information om statusuppdateringar från klassen vi är intresserad av (kallas observable). Klienter kan sedan implementera interfacet och registrera sig för att lyssna.

Vilka designproblem löser man genom att använda detta pattern? (X)

I MVC behöver *View* känna till förändringar som sker i *Model* . Lösningen på en sådan situation kan t.ex. vara "polling" (där V regelbunden frågar M om status). Observer Pattern löser situationen på ett smidigare sätt där M inte beror av V och där det är enkelt att utöka med fler *Views* (OCP).

Template Method

När kod som till stora delar är gemensam men beror på en liten del som inte är det:
Bryt ut det som är gemensamt i en abstrakt klass, och låt det som inte är gemensamt representeras av en abstrakt metod som kan implementeras olika i subklasserna (fungerar även med delegering och interfaces).
Det ska finnas en templateMethod i den abstrakta klassen som innehåller de abstrakta metoderna.

När kan man använda det?
När man har kod som till stora delar är gemensam och vill återanvända den.

Hur kan man använda det?
Det som står under Template Method.

Vilka designproblem löser man genom att använda detta pattern? (X)
Duplicerad kod.

Strategy

När en mestadel generisk algoritm kan variera i detaljerna:
Skapa ett interface med metoder som representerar olikheterna och använd metoderna inuti den generiska algoritmen. Definiera olika konkreta beteenden i separata klasser som implementerar interfacet och lämnar över dem till den generiska algoritmen när nödvändigt.

När kan man använda det?
När du vill kunna välja mellan en av många strategier och sedan köra din algoritm med den.

Hur kan man använda det?
Det som står under Strategy.
Implementeras på samma sätt som State- och Bridge Pattern, men sättet vi använder dem skiljer sig åt.
Ex: En bil kan köra på olika sätt, t.ex. så fort eller så bränslesnålt som möjligt. Funktionen att "köra" kan, vid olika tillfällen, ta en sådan strategi som argument och anpassa beteendet därefter.

Vilka designproblem löser man genom att använda detta pattern? (X)
Istället för att ha flera separata objekt där endast beteende skiljer dem åt, kan man ändra beteende internt i ett enda objekt m.h.a. olika strategier.

State

När ett objekt kan variera mellan olika tillstånd:
Skapa ett interface med metoder som representerar olikheterna och använd metoderna inuti objektet.
Definiera olika konkreta tillstånd i separata klasser som implementerar interfacet och använder dem internt i objektet när nödvändigt.

När kan man använda det?
När man vill kunna byta mellan olika tillstånd över tid. Kombineras gärna med Singleton Pattern då det bara bör finnas ett objekt som representerar varje specifikt tillstånd.

Hur kan man använda det?
Det som står under State.
Implementeras på samma sätt som Strategy- och Bridge Pattern, men sättet vi använder dem skiljer sig åt.
Ex: En amfibiebil kan växla mellan att köra på land eller vatten. Den har samma uppsättning beteenden (köra, svänga, backa, etc), men de implementeras olika för varje tillstånd (land / vatten).

Vilka designproblem löser man genom att använda detta pattern? (X)
Istället för att ha flera separata objekt där endast tillstånd skiljer dem åt, kan man ändra tillstånd internt i ett enda objekt m.h.a. olika tillstånd.

Structural patterns

Berör hur vi strukturerar komponenter (vilka klasser & interfaces vi bör använda och hur de bör bero av varandra).

Bridge

När en aspekt av ett objekt kan variera oberoende av objektet självt:
Skapa ett interface med metoder som representerar aspektet och använd metoderna inuti objektet.
Definiera olika konkreta implementationer av aspektet, och använd dem med objektet när nödvändigt.

När kan man använda det?
När vi vill veta hur vi bör tänka för att åstadkomma mesta möjliga variabilitet och polymorfism för objekt bestående av olika komponenter.

Hur kan man använda det?
Det som står under Bridge.
Implementeras på samma sätt som Strategy- och State Pattern, men sättet vi använder dem skiljer sig åt.
Ex: Det finns olika sorters bilar och motorer. Vi kan sätta ihop en motor med en bil och få en specifik kombination, men bilar och motorer kan variera oberoende av varandra.

Vilka designproblem löser man genom att använda detta pattern? (X)
Inflexibilitet. Man vill kunna byta ut komponenter oberoende av varandra.

Adapter (wrapper)

Möjligheten att göra annars inkompatibla komponenter kompatibla med varandra.

När kan man använda det?

När kod som tidigare fungerat med komponent X nu också behöver fungera med komponent Y som har ett annat gränssnitt.

Hur kan man använda det?

Man skapar en adapter-klass som hanterar konvertering och anpassar sig efter klientens gränssnitt så att den innehåller det klienten förväntar sig.

Vilka designproblem löser man genom att använda detta pattern? (X)

Vi slipper skapa ny kod med samma funktionalitet, där det enda som skiljer är typen.

Composite

Gör det möjligt att hantera grupper av objekt som om de vore ett enskilt objekt av den typen.

När kan man använda det?

När man använder många objekt på samma sätt och ofta hanterar dem med nästan identisk kod.

Hur kan man använda det?

Skapa composite-objekt som ändrar en grupp av objekt (*Leaf* / *Löv*) som är kopplade till den. Då behöver man endast ändra composite-objektet för att ändra hela gruppen.

Vilka designproblem löser man genom att använda detta pattern? (X)

Repitativ kod.

Decorator

Attach additional responsibilities to an object dynamically as separate adapter objects, keeping the same interface.

När kan man använda det?

När man har en komplex klass som inte följer SRP, och för att lösa detta vill man skapa en bas-klass som kan lägga till funktionalitet *dynamiskt* (under run-time).
Ett exempel på detta är en pizza: du vill ju inte ha en pizza som innehåller ALLA ingredienser! Du börjar med brödet (bas-klassen) och addera de ingredienser du faktiska vill ha.

Hur kan man använda det?

Skapa en bas-klass samt andra klasser som inrymmer de beteenden bas-klassen vill implementer *dynamiskt* .
Alla klasser implementerar ett gemensamt interface.

Vilka designproblem löser man genom att använda detta pattern? (X)

När objekt blir så komplicerade att SRP inte följs och har ett behov att ändras *dynamiskt*.

Facade

Hide internal complexity by introducing an object that provides a simpler interface for clients to use.

När kan man använda det?

När man vill tillhandahålla ett enklare gränssnitt till sina klienter, reducera klientens beroende på interna detaljer, eller kanske till och med dölja ett dåligt designat gränssnitt bakom ett bättre.
Facade Pattern kan användas som ett "skyltfönster" som tillhandahåller de vanligaste operationerna.

Hur kan man använda det?

Skapa en klass eller ett interface med ett förenklat gränssnitt för ens klienter att använda. Exempelvis kan klassen Factory fungera som en Facade.

Vilka designproblem löser man genom att använda detta pattern? (X)

När den interna strukturen är för komplex för en klient att hantera smidigt och enkelt.

Architectural patterns

Berör hur vi strukturerar våra program på en högre nivå än klasser.

Module

Gruppera flera relaterade element (klasser, interfaces, metoder...) som en konceptuell enhet.

När kan man använda det?

När det finns flera relaterade element och man vill kunna utnyttja *package-private* istället för t.ex. *public* .

Hur kan man använda det?

Förr användes en klass som representerade "modulen" genom att inrymma hela beteendet.
Nu skapar man bara en mapp (module) och placerar alla relaterade element i den.

Vilka designproblem löser man genom att använda detta pattern? (X)

När kodstrukturen är för komplex. Det löser också problemet med att mycket kod har *public* som *access modifier* för att kunna nås, det leder till onödig och riskabel exponering.

Model-View-Controller (MVC)

Ensure that the model (M) does not depend on the user interfaces (V & C).

Model (smart): Hanterar data, tillstånd, domänlogik, och är en representation av den domän som programmet arbetar över. M FÅR INTE bero av V eller C ("The whole model and nothing but the model.")!
Smart: All domänlogik ligger i modellen.

View (dumb): Beskriver modellen för användaren i form av t.ex. text, grafik och ljud. Vi vill (oftast) att V uppdateras när M den presenterar uppdateras.
Dumb: V ska inte utföra några egna beräkningar. Den ska bara "visa" utifrån direktiv.

Controller (thin): Styr M utifrån input från användaren (direkt användarinput, rörelsesensorer, etc).
Thin: C ska enbart hantera yttre input och är därför bara ett tunt lager mellan användare och program.

När kan man använda det?

När ett program innehåller någon form av "grafisk" representation.

Hur kan man använda det?

Grunden i MVC är att separera M från användargränssnittet (V & C). Inom användargränssnittet skiljer vi kod som visar upp modellen för användaren (V) från kod som hanterar input från användaren (C). Vad gäller beroenden är det enda viktiga att M inte får bero av V eller C. Tumregel för resterande beroenden är att göra det som leder till högst cohesion och lägst coupling.

Vilka designproblem löser man genom att använda detta pattern? (X)

Då koden blir uppdelad i M, V och C kan flera programmerare koda samtidigt utan att störa varandra. Eftersom koden är avskild är den också lättare att återanvända i andra applikationer, samt att den ger hög cohesion och låg coupling.

Grundläggande OO koncept

Klass
* En kod-enhet som definierar data (tillstånd) samt operationer på datan.
* Definierar en typ.
* Är ett statiskt begrepp.

Objekt
* En instans av en klass (eller vektor).
* Sparar värden av vissa typer.
* Är ett dynamiskt begrepp.
* Är superklass till alla klasser i Java.

Konstruktor
* Speciell metod som används för att skapa objekt och initialisera instansvariabler.
* Har alltid samma namn som klassen.
* Har ingen returtyp (inte ens void).
* En klass kan ha flera konstruktorer med olika parametrar.
* Om man inte definierar en egen konstruktor finns det en defaultkonstruktor.
* Anropas (implicit) när ett *new* -uttryck evalueras.

Abstrakt klass
* En klass som inte kan skapa några objekt.
* Syftet är att samla gemensam kod för subklasser.
* Kan ha abstrakta metoder (metoder utan body) som subklasser måste göra override på, om de inte själva gör dem abstrakta.
* Konstruktorn kan enbart anropas via *this()* i den egna klassen eller *super()* i subklasserna.

Interface
* En specifikation med ett antal metod-signaturer som tillsammans bildar en typ.
* Enbart signaturer ges – samtliga metoder är abstrakta.
* Interfaces i Java får specificera "konstanter", dvs attribut som implicit är *public static final* , och som initialiseras till ett konstant värde. Används dock sällan.

Subklass
En klass som ärver av sin superklass. Det enda som inte ärvs är superklassens konstruktor.

Implementationsarv
* Deklarerar att klass A ärver av klass B.
* Alla komponenter från B är också (implicit) komponenter av klass A (förutom konstruktorer som ej ärvs).
* B är en superklass av A och omvänt är A en subklass av B.

Specifikationsarv
* Deklarerar att klass A implementerar ett interface B.

Primitiv typ
Är den typ som värden i Java har.

Namn	**Längd**
byte	8-bit integer (-128 / 127)
short	16-bit integer (-32 768 / 32 767)
int	32-bit integer (-2^{31} / 2^{31}-1)
long	64-bit integer (-2^{63} / 2^{63}-1)
float	32-bit floating point
double	64-bit floating point
char	16-bit unicode character
boolean	32-bit, *true* eller *false*

referens / pointer (små och enkla värden) är adresser till en minnesarea där objekten bor.

Referenstyp
* Alla typer som inte är primitiva kallas referenstyper.
* I Java finns det två sorters referenstyper – array-typer samt klass- eller interface-typer.
* String, Object, Runnable, char[], ArrayList<Polygon> är exempel på referenstyper.
* Alla referenstyper har värden av samma sort: referenser.
* Typen för en referens används för att garantera att referensen i fråga pekar på ett objekt som kan uppföra sig som förväntat, dvs har de metoder och attribut som förväntas av typen.

Alias
Två variabler eller attribut som håller samma referensvärde, dvs pekar till samma objekt.

Instansvariabel / Attribut
* Representerar en del av objektets tillstånd.
* Är synlig i klassens metoder eller scope.
* När man skapar ett objekt initialiseras instansvariabler (implicit) till defaultvärden beroende på typ.

Referensvariabel
Refererar till ett objekt på heapen.

Statisk typ
En typ som bestäms under *kompilering* och kommer vara densamma när programmet körs. Fördelen med statiska typer är att man kan upptäcka fel tidigt och åtgärda dem.

Dynamisk typ
En typ som bestäms under *exekvering* (runtime-polymorfism).

Statisk metod & Statisk variabel
* En klassmetod och klassvariabel kan nås utan ett instansobjekt.
* Anropas m.h.a. namnet på klassen, t.ex. *ClassName.staticMethod* och *ClassName.staticVariable* .

Icke-statisk metod
* Metod som nås via ett instansobjekt.
* Anropas m.h.a. namnet på objektet, t.ex. *objectName.length();*

Inkapsling
* En av de fyra fundamentala koncept inom OOP (tillsammans med arv, polymorfism och abstraktion).
* Syftet är att gömma information (**data hiding**) från andra klasser än sin egen m.h.a. *access modifiers* : public, protected, package-private och private.
Klasser: Kan sättas till *public* eller *package-private* .
Metoder & attribut: Kan sättas till *public* , *protected* , *package-private* eller *private* .
Variabler: Deklarera som *private* och tilldela getters & setters för att modifiera och se dess värde.

Abstraktion (kommer ej på muntan)
Handlar om att exponera så lite information som möjligt för att skapa lösa beroenden.
Ju mindre en klass exponerar (metoder, attribut...) desto mindre finns det för andra att bero på.

Abstraktion vs Inkapsling (kommer ej på muntan)
"You do abstraction when deciding what to implement.
You do encapsulation when hiding something that you have implemented."

Overloading
* Ett objekt som har flera metoder med samma namn, men olika signaturer.
* Vilken av signaturerna som används bestäms *statiskt* (vid kompillering).

Overriding
* En subklass kan definiera en metod med samma namn och signatur som en metod i sin superklass.
* Vilken av implementationerna som används bestäms *dynamiskt* (vid exekvering).
* Vi säger att metoden är *polymorf* .

Dynamisk bindning *(dynamic dispatch i detta sammanhang)*
Innebär att ett val mellan flera möjliga implementationer av en polymorf metod avgörs vid run-time.

Tekniker

Functional decomposition (kommer ej på muntan)

När man bryter ner ett program i delar för att reducera dess komplexitet.
Hur: Enklare delberäkningar bryts ut till egna metoder som anropas från den ursprungliga metoden.
Fördelar: Möjliggör kodåteranvändning och testbarhet för de delberäkningar man brutit ut och det reducerar komplexitet för varje metod. Bra namngivning ökar förståelsen.

Refactoring

Omstrukturering av kod som bevarar funktionalitet men förbättrar struktur.

Method Chaining (kommer ej på muntan)

Den språkfeature som låter oss sätta ihop flera metodanrop i en "kedja" utan att introducera lokala variabler för mellanstegen, t.ex. *myPoly.translate(10, 10).rotate(Math.pi).scale(2, 2);*

Method Cascading

En specifik användning av *Method Chaining* där objektet självt returneras från varje anrop i kedjan.
En smidig teknik är att låta alla mutators returnera *this* istället för *void* .

Defensive Copying

För att garantera att **inga alias** till eventuella muterbara objekt finns, behöver vi skapa *defensive copies* (kopia av objektet returneras, inte referensen till den) om:
* Vi exponerar objekten via **getters**.
* Vi i instansmetoder skickar objekten som argument till **externa metoder** (de kan ha sidoeffekter).
* Vi tar in nya värden för attributen via konstruktorer eller setters.
OBS! Detta används inte bara när vi strävar efter immutabily. Vi vill skydda oss mot oväntade förändringar via alias.

Mutate-by-copy

Istället för att **ändra objektets tillstånd**, skapa en kopia med det nya tillståndet och returnera den istället.
Då det ursprungliga objektet inte ändras kan vi inte bli överraskade av *alias updates* , men det kan leda till att många nya objekt skapas vilket kan bli ineffektivt.

Avancerade språkmekanismer

Exceptions

* Moderna imperativa programspråk (som Java) har strukturerad felhantering via *exceptions* . Detta innebär att vi slipper använda *felkoder* som ökar komplexiteten. Använd **aldrig** felkoder i Java.
* Ett exception i Java är ett objekt som representerar och innehåller information om ett fel som uppstått.
* Alla former av "felobjekt" i Java är subklasser till Throwable och kan kastas (**throw**) eller fångas (**catch**).
* När ett exception inträffar innebär det en form av non-local transfer of control. Koden följer inte den normala strukturen och kan "hoppa" till ett catch-block långt bort där exception kastas.
* *Error* representerar fel som inte går att återhämta sig från, exekvering ska avslutas.
* Unchecked- vs Checked exceptions:

Unchecked: RuntimeException representerar "buggar", saker som inte borde inträffa och därför inte borde varnas för. T.ex. *ArrayIndexOutOfBoundsException* , *IllegalArgumentException* och även *Error* .
Checked: Alla andra exceptions. De representerar saker som förväntas kunna inträffa under normal körning. T.ex. *FileNotFoundException* och *SQLException* .

Mutability

* Objekt i Java är som standard *muterbara* , dvs kan förändras.
* Den enklaste form av förändring är att värdet på attribut ändras (via setters).
* Objekt kan även vara muterbara för attribut som håller referenser till andra muterbara objekt, då en förändring av dessa också implicit innebär en förändring av objektet som håller referensen. T.ex. Om objekt A har ett attribut som är en array, och objekt B har tillgång till en referens i samma array. Då kan B uppdatera innehållet i arrayen och därigenom förändra tillståndet för A.

Immutability

* Ett icke muterbart objekt är ett objekt vars tillstånd inte kan förändras efter att det skapats.
* Objekt bör vara immutable som standard. Låt endast objekt vara muterbara om deras syfte är att ofta ändras.
* Behöver ett objekt vara muterbart får man se till inga alias-problem uppstår (Defensive Copying) och att objektet internt beror på immutable objects så mycket som möjligt.

"Classes should be immutable unless there's a very good reason to make them mutable. If a class cannot be made immutable, limit its mutability as much as possible" – Joshua Bloch

* Immutable objekt är automatiskt *trådsäkra* och lämpar sig bra för parallella beräkningar.

Trådar

* I Java beskriver man aktiva objekt (och parallellism) m.h.a. klassen *Thread*. Aktiviteter som pågår samtidigt kallas därför i Java för trådar.
* I ett parallellt program beskrivs varje aktivitet som en instans av klassen *Thread*. Det tråden ska utföra definieras genom att override:a metoden *run()*.
* För att starta exekveringen av en tråd anropar man metoden *start()*. Anropas *run()* direkt skapas ingen tråd.
* Program med parallellism har ett icke-deterministiskt beteende: i vilken ordning operationerna sker mellan de olika programflödena vet vi inte, och två program med samma indata kan ge olika resultat.
* En tråd kan temporärt avbryta sin exekvering med metoden *sleep()*. Den kommer i två varianter, en som avbryter trådens exekvering i millisekunder och en som gör det i milli- **och** nanosekunder.
Eftersom *sleep()* kan kasta en kontrollerad exceptionell händelse måste anropet ligga i ett try-catch-block.
* Java tillåter inte multipla implementationsarv och därför kan det bli problem att ärva från *Thread*. Lösningen är att använda interfacet *Runnable* som endast innehåller metoden *run()*.

Trådsäkerhet

* Då ett objekt modifieras kan den anta ett antal tillfälliga tillstånd som inte är konsistenta / giltiga.
* Om en tråd avbryts under modifiering av ett objekt kan det lämna objektet i ett ogiltigt tillstånd.
* En klass sägs vara trådsäker om den garanterar konsistens för sina objekt även om det finns multipla trådar.
* *Kritiska sektioner* är kodsegment som har access till samma objekt från olika trådar. Detta måste synkroniseras så att endast en tråd i taget får tillgång till objektet, annars kan objektet hamna i ett ogiltigt tillstånd.
* *Race condition* är när en tråd försöker läsa data samtidigt som en annan tråd uppdaterar den.
* För att ett *race condition* inte ska uppstå måste *kritiska sektioner* synkroniseras med *synchronized*.
* Synkronisering handlar om uteslutning, inte samarbete.

Munta-specifika frågor

Frågor om centrala begrepp

1.1 Vad är polymorfism?

Att ett objekt av en viss typ kan användas som om den vore ett objekt av en annan typ.

1.2 Vad är subtypspolymorfism?

Ett objekt av en subtyp kan agera som om den vore ett objekt av sin supertyp (superklass eller interface).

1.3 Vad är parametrisk polymorfism?

Generics. Förmågan att skriva funktioner och datatyper generiskt så att de kan hantera flera olika sorters typer utifrån användandet.

2. På vilket sätt blir kod bättre av att man använder parametrisk resp. subtypspolymorfism? (X)

* Subtypspolymorfism: Åstadkommer code reuse via arv, eller implementation av interface via delegering.
* Parametrisk polymorfism: Åstadkommer code reuse genom att återanvända kodstrukturen. Du kan t.ex. ha en generisk metod istället för flera likadana, där det enda som skiljer dem åt är vilken typ de hanterar.

3.1 Vad är kovarians?

Ett praktiskt tillvägagångssätt om man endast vill kunna läsa från en lista, t.ex. arrays.
Exempel: List<? Extends Polygon> cov = new ArrayList<Triangle>();
* Vi kan läsa element ur listan, för vad ? än representerar vet vi att objekten kan betraktas som en *Polygon*.
* Vi kan inte skriva till listan. Antag att vi vill lägga till en *Triangle*, hur vet vi att det inte är en lista av *Square*?

3.2 Vad är kontravarians?

Ett praktiskt tillvägagångssätt om man endast vill kunna skriva till en lista.
Exempel: List<? super Polygon> contv = new ArrayList<Polygon>();
* Vi kan faktiskt läsa från listan, men bara som typen *Object* eftersom inget kan existera som inte är en subtyp till *Object*.
* Vi kan lägga till element av typen *Polygon* eller dess subtyper till listan. Vad ? än representerar vet vi att det är en supertyp till *Polygon*.

4.1 Vad är delegering?

När en klass håller en referensvariabel till en annan klass. Detta används för att nå den andra klassens metoder utan att behöva ärva.

4.2 Vad är arv?

En mekanism för återanvändning av kod.

5. Hur åstadkommer man kodåteranvändning med delegering resp. arv?

Delegering: Man skapar ett objekt av en annan klass och når dess metoder via objektet.
Arv: Klass A ärver (extends) av klass B. Allt i klass B (förutom dess konstruktorer) kan nu användas av A.

6.1 När bör man använda arv? (X)
Följ LSP (finns i avsnittet Design-principer).

6.2 Varför säger vi att man kan föredra komposition framför arv (Composition Over Inheritance)? (X)
Läs *Composition Over Inheritance* (finns i avsnittet Design Patterns).

Frågor av mer teknisk eller Java-specifik karaktär

7.1 Vad är en typkonstruktor?
* Arrays: [] kan ses som en typkonstruktor för arrays. Vi har ingen faktskt typ förrän vi anger typen för elementet, t.ex. String[].
* Generic types: ArrayList<_> är en typkonstruktor. Vi måse ange en argumenttyp för att kunna skapa objekt av en specifik typ, t.ex. ArrayList<String>.

7.2 Vad är en typparameter?
* En variabel (T) som representerar en typ vi inte känner till. Den bestäms av det argument som ges vid instansiering och kan variera för de olika gånger den instansieras.
* En typparameter kan använda sig av **upper bounds** (*public class ClassName< **P extends Polygon** > { ... }*), vilket ger oss möjlighet att kontrollera vilka sorts typer vi kan ge som argument.

8.1 Vad är en typvariabel?
I *public <T> void myMethod(T name) { ... }* är *name* typvariabeln.

8.2 Vad är ett wildcard?
Ett typargument (?) som representerar en okänd typ. Vi kommer aldrig veta mer om denna typ än de bounds (upper och lower) som är givna.

9. Vilken utsträckning tillåter Java varians vid metod-overriding?
Metoder är covarianta i sin returtyp och invarianta i sina argumenttyper. Java tillåter inte contravarians som argumentyp, den betraktar det som *overloading* om vi varierar argumenttypen åt något håll.

10. Vad innebär nyckelordet extends och super?
Extends: Klasser som använder *extends* bredvid klassdeklarationen ärver från den klass de specificerar.
Super: Ett sätt för en subklass att anropa objekt i sin superklass.

11. När bör man använda extends och super? (X)
Extends: När man vill ärva.
Super: När en subklass behöver:
* Nå variabler i superklassen vars namn är samma som variabler i subklassen.
* Anropa superklassens konstruktor.
* Anropa superklassens metoder som subklassen har gjort *override* på. Då har man tillgång till båda metoderna, t.ex. *runMethod(...){...}* (subklassens metod) och *super.runMethod(...){...}* (superklassen metod).