2.1 **Redogöra för objektorienterad design-principer för några av SOLID-principerna, samt för någon av de övriga principerna vi tagit upp under kursen:**

- ✓ När de fem SOLID principerna är kombinerade gör de det lättare för en programmerare att programmera kod som är enkel att förstå, göra den flexibel, samt underhålla.

1. ***(1)Vad säger principen? Ge något konkret exempel på hur man bryter mot eller, följer, principen. (V) Vad fyller principen för syfte?***
   (2)Varför är det bra att följa principen?

S - Single Responsibility Principle: (1) En enda modul eller klass ska bara ha ett enda ansvarsområde, annars delar du upp den i flera olika klasser/moduler.
Anledningen till att det är viktigt att hålla en klass inriktad på ett enda bekymmer är att det gör klassen mer robust.

(2) Om vi till exempel har en modul som både kompilerar och printar en uppsats, då har modulen 2 anledningar till att ändra på sig, innehållet av uppsatsen kan ändras, formatet kan ändras. Det hade istället varit bättre att bryta upp modulen och låta en del sköta printen och en annan modul sköta kompileringen.

O - Open Closed Principle: (1) Klasser, moduler, funktioner etc. skall vara öppen för tillägg, men stängd för modifikationer. Vad det menas med är att kod bör ej förändras varje gång kraven förändras. Ett exempel på hur man bryter mot den är om vi har en klass *Rectangle* och en klass *AreaCalc* som ska räkna ut arean av en figur. Nu kan vår *AreaCalc* räkna ut arean av en *Rectangle* men om vi vill lägga till att den även ska lägga till arean av en *Circle* kan vi göra det i *AreaCalc*... men då har vi gjort modifikationer i koden när syftet med OCP är att man ska lägga till och inte göra några modifikationer. På så sätt är det på bästa sätt att skapa en klass *Shape* som innehåller en *Area()* metod som går att overrida för varje distinkt klass som använder sig av den som då ärver från *Shape* klassen.

(2) Den tillåter massa komplikationer i kodens struktur samt att det gör koden enklare att förstå när vem som helst kollar på den. Det bästa alternativet när man vill "ändra" en kod är att skapa en ny klass istället för att ändra den och på så sätt även ta bort de klasser som ej används.

L - Liskov Substitution Principle: (1) Om klass *S* är en subtyp av klass *T* då kan objekten av typ *T* ersättas med objekten av typ *S*.

Om vi har en klass *Bird* och subklasser som *Struts*, eftersom att *Struts* inte kan flyga så kommer det inte att gå när man byter ut *Bird* till *Struts*, flyga metoden kommer göra så att koden crashar.

Istället ska de vara så att om vi har en klass bil och en subklass volvo240, ska det gå att byta ut bil till volvo240 i programmet och koden ska fungera felfritt.

(2) Kod som följer *LSP* är löst beroende av varandra och uppmuntrar återanvändning av kod. Använder basklassen ofta som en utgångspunkt för att skapa någonting nytt istället för att repetera kod.

I - Interface segregation Principle: (1) Säger att ingen klient bör ej bli tvingad till att beroende av en metod som den ej använder. Därför delar man upp interfacen till små delar där de endast är beroende av de metoder som är av betydelse. Små interfaces är lättare att referera till. Fördelen med att skapa små interfaces och att separera dem till olika roller och ansvarsområden är att då gynnar vi composition(sammansättning) över inheritance(arv) och decoupling över coupling.
Exempelvis en app har många olika "kategorier" och istället för att samla alla i en interface delar man upp kategorierna i små interfaces och sammanställer dem i de syfte vi vill att de ska utövas på.

(2) *ISP* är avsedd att hålla ett system frikopplat och sålunda lättare att refactor, ändra och omplacera kod.

D - Dependency inversion principle: (1) Den säger att man ska invertera beroenden så att high-level moduler ej är beroende av low-level moduler utan bero på abstraktioner och inte på konkreta implementationer.

- Minimera beroenden
- ha så lösa beroenden som möjligt
- ha kontroll över beroendena
- varje beroende ska vara avsiktliga
- kopplingspunkter (möjlighet att skapa beroenden) ska vara avsiktliga
- kopplingspunkter ska ha väldefinierade gränssnitt

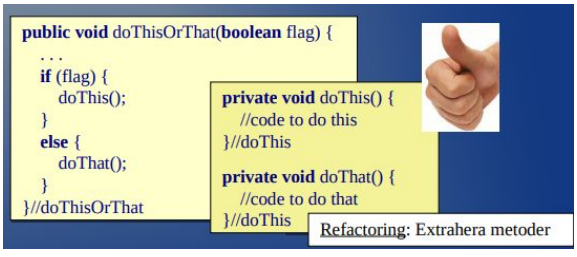
Tex: Observer i MVC som vänder på beroendet från modellen till viewen, så att det istället pekar på ett interface(observern)

(2) DIP kan medföra mer komplex kod men den utökar kodens flexibilitet.
Template design pattern är ett exempel där DIP tillämpas.

High Cohesion - Low Coupling:
(1) HC Principen hänvisar till att man ska ha hög sammanhållning och binda all relaterad kod tillsammans så nära som möjligt. Ett sätt att se på sammanhållning när det gäller OO är om metoderna i klassen använder någon av de privata attributen.
(1) Loose eller low coupling föreslår att klasser ska ha så lite beroenden som möjligt, och beroenden bör vara svaga beroenden om de nu måste existera, detta uppnås helst genom interfaces.

(2) High cohesion och low coupling ger oss bättre utformad kod som är lättare att underhålla.

Separation of Concern: (1) En metod skall endast göra en sak och göra denna sak bra! Exempelvis om vi har en metod doThis och doThat i en och samma metod så kan det uppstå onödiga komplikationer och man kan ej återanvända koden på ett vettigt sätt, därför är det bäst att extrahera metoderna så att varje metod endast gör en sak.



(2) Följer man Separation of Concern fås små och överblickbara metoder som är lätta att läsa och förstå!

Invarians:
Ex: List<Polygon> l = new ArraList<Polygon>();
När du vill både hämta och lägga till.
Kovarians:
Ex: List<? extends Polygon> l = new ArraList<Triangle>();
När du vill hämta något. Du är säker på att ? är en Polygon. Du kan inte lägga in något eftersom du inte vet vad ? är.

en metod som gör @Override kan t ex returnera en Square om motsvarande metod i superklassen returnerar Polygon

Kontravarians
Ex: List<? super Triangle> l = new ArraList<Polygon>();
När du vill lägga till något.

overriding ej möjligt, blir overloading.

Get-Put Principle: *(1)* När du **get**(hämtar) objekt ur en struktur använd (covarians) **extends** wildcard. När du **put**(placerar) objekt i en struktur använd (contravarians) **super** wildcard. När du gör bådadera, använd (invarians, ingen förändring) inte wildcard.

Ett undantag är att du inte kan **put** någonting i en typ deklaration som har **extends** wildcard förutom värdet ***null***, för att den tillhör varje referens typ.

Ett annat undantag är att du inte kan **get** någonting i en typ deklaration som har super wildcard förutom typer av ***objekt***, för att den är en super typ av varje referens typ.

Ett exempel är om vi har en kollektion av bananer. vi har Collection<? extends Fruit> vilket är en kollektion av frukter men vilka typer av frukter vet vi ej, man kan hämta en frukt från den, däremot vet vi inte vilken typ av frukt vi kan få. Om vi lägger till äpplen till en kollektion av ananas, vilket lär generera ett fel. Kan endast lägga till ***null*** till det.

Om vi har en fruktskål. vi har Collection<? super Banana> i den kollektionen finns det någon typ som är större än Banana . I denna kollektionen kan vi lägga till Banana men om vi försöker hämta något från den kan det mycket väl vara något annat än en Banana.

(2) Get - put principen säger till oss att typen är inte densamma om vi lägger in en typ som inte är en subtyp av värdet.

Law of Demeter: (1)Säger att vi ska reducera beroenden hos klasser, och endast ha kontakt med våra direkta grannar för att undvika method chaining när vi vill komma i kontakt med andra objekt.

Ett exempel på hur vi bryter mot LoD är om vi har en metod person och dess funktion säger person.getName som då kallar på namnet som är i någon annans objekts instansvariabel vilket bryter mot LoD.

Alltså för att följa LoD får vi kalla på oss själva(*this*), metodens argument objekt, instansvariabler i objekt av **klassen**, objekt som är skapade av **metoden** eller genom

funktioner eller metoder som **metoden** kallar på och objekt i globala variabler (*static*) räknas som argument av **metoden**.

Klasser ska bara ha beroenden till klasser som är oundvikliga, dennes vänner. Ska bara anropa metoder hos sina vänner. För att uppnå detta kan man tex använda factory patterns eller observator patterns när man måste kontakta klasser utanför vänskapen.

För att sätta det i ett exempel; En kusk kan styra sina hästar och kan frakta passagerare. Passagerarna ska inte styra hästen och kusken kan prata med passagerarna.

Command-Query Separation: En metod ska antingen ha en sidoeffekt (mutator) eller returnera ett värde (accessor), och inte båda.

- En metod som returnerar information om ett objekt, ska inte ändra på objektets tillstånd.
- En metod som ändrar tillståndet på ett objekt, ska inte returnera information om objektet.

Någonting som bryter mot detta är då incrementSpeedAndReturnSpeed();

2.2 Design-mönster, samt deras syfte och effekt

1. **Vad är ett design pattern?**
2. **(V) Varför använder man design patterns?**

---

1. Ett tillvägagångssätt för att strukturera kod för att (ofta) uppnå design principles.

2. Design patterns är lätta att känna igen och de har lösningar till vanliga problem. Samt att de är lätta att underhålla eftersom många är bekanta med dem.

Template Method: bryt ut kod som är gemensam till en abstrakt klass. Skapa abstrakt metod i den abstrakta klassen för det som inte är gemensamt, så att subklasserna kan implementera sin egen version av den. Subklasser overridar existerande metoder av template klassen.

principles: Interface Segregation Principle

Bridge: När ett objekt är sammansatt av flera olika komponenter som kan variera oberoende av huvud objektet, ska man skapa separata interface för komponenterna och låta dem implementera dessa.
Tex: Kombinera olika bilar och motorer på olika sätt

Factory Method: Ett mönster för att minska beroenden vid instansiering. Om man vill skapa objekt kan man använda en factory (statisk klass som innehåller metoder som returnerar objekt man vill skapa) som skapar det åt en. På så sätt får man inte ett direkt beroende till objektet man skapar.
Principles: Dependency Inversion Principle

Abstract Factory: Ha en abstrakt factory som använder sig av flera konkreta factory som arbetar med samma objekt fast med små skillnader. På så sätt kan användaren välja vilken hen vill ha.
Tex: Factory: fourDoorVolvoFactory...

Facade: Ett sätt att gömma utomstående funktionalitet genom att skapa en fasad som har tillgång till funktionaliteten men som inte visar den och visar en förenklad vy av den vilket gör det enkelt att använda. Factory är en slags fasad.

Chain of Responsibility: När ett anrop görs vill vi att det ska se ut som att det görs via ETT objekt men i själva verket kan det skickas via flera objekt innan det behandlas. Detta kan göras med ett interface. Du kan modifiera existerande funktionalitet, likt när man overridar en metod. Kan kalla på super.x(); för att fortsätta upp i kedjan eller hantera meddelandet själv.

Ex: Skicka post, Post -> ? -> Goal

Module: på hög nivå gruppera gemensamma metoder/klasser/paket/moduler för att åstadkomma struktur.

Observer: Det löser problemet med att observatören har direkt kontakt med observatörerna. Ha en observer (interface) som lyssnar på saker som händer och broadcastar det sen till de som observerar.

Till exempel om kontakten med observatörerna har en massa händelser skulle den behöva att ge var och en av observatörerna information om händelsen. Så med Observer pattern kan vi lösa en-till många förhållandet. Det vi gör är att observatörerna registrerar sig för observerbara så att de kan få meddelande om nya händelser så att alla kontaktar en modul i stället för en modul kontaktar var och en.

State: Tillåter objekt att bete sig på olika sätt beroende på det interna tillståndet. Skapa interface som pekar på olika state-klasser beroende på vilket tillstånd koden befinner sig i. I klasserna finns de metoderna som är specifika för just den staten. Om du har olika states så kan du välja dem så slipper du flytta dem till ett annat ställe för att de ska bli executade.

Tex: I StateDriveable fungerar gas som vanligt. I StateUndriveable händer inget i gas. IDrivable driveable = new StateDriveable();

Strategy: typ samma som state pattern fast med olika strategier, olika sätt att utföra en viss uppgift på. Jag använde de oftast när jag hade en massa if-else statements, så för att minska dem använde jag strategies.

Tex: En bil kan köras på olika sätt, du har då olika strategies som GasStrategy, StopStrategy etc som kan användas när vi byter strategies.

Decorator: Omsluter redan existerande kod med ny funktionalitet. Görs med en abstrakt decorator-klass och konkreta decorator-klasser som specificerar den utökade funktionaliteten. Metoder anropas därefter från dessa, som skapas som instanser, om den nya funk vill användas.

Tex: Sandwich macka = new Ham(new Cheese(new Tomato));

Iterator: Användning av denna design möjliggör avskärmning av intern funktionalitet. Detta leder till att användare kan bruka, till exempel, for-each-loopar för att få access till element i objekt utan att behöva bry sig om dess underliggande representationer.

Visitor: Ett sätt att använda funktionell programmering i oo-programmering. I oo kan nya objekt läggas till utan att ändra på existerande kod. I funk kan ”metoder” läggas till utan att ändra på existerande kod. Via visitor kan man i oo lägga till fler metoder till objekt vid runtime utan några problem. Detta leder dock till att man inte kan lägga till fler objekt.

Ex: En kund beställer en taxi, som anländer vid dörren. När kunden sätter sig i taxin så är taxi besökaren i kontroll av transporten för kurden.

Summary: Separera viss logik från elementen, vilket gör data klassen enkel.

Adapter: Sammankopplar två interfaces som inte är kompatibla. Detta görs med hjälp av en ensam klass som slår ihop funktionalitet hos de båda interfacen.

Singleton: Den säger att en klass bara har en instans som kan bli tillgänglig globalt. Man kan bara skapa en instans av ett objekt och det är bara det som returneras när det

efterfrågas. För att åstadkomma detta kan man göra konstruktorn private och använda sig av en (static) getter som bara returnerar det objektet.

Dåligt att använda sig av den då vi antar att det aldrig behövs mer än en instans av ett objekt. Exempelvis i messenger har vi en gruppchatt men med tiden vill vi ha en till.

Composite: används när vi vill behandla flera objekt som ett objekt. Children sparas i en lista som sedan beter sig samma som huvud objektet.

Model-View-Controller: Är till för att hålla isär roller/ansvarsområden hos objekt. Meningen med en korrekt implementerad MVC-design är att skapa en kod som går att representera på olika sätt utan att modifiera själva modellen.

Till exempel övergång från 2D till 3D.
Eller myra och en myr koloni, en myra springer runt hela dagen och samlar mat, dens roll är annorlunda från de resterande myrorna.

- Model : the data and its business logic
- View : representation of the data
- Controller : coordination between the model and the view.

Principal: Separation of concern.

Command: Motivationen till att använda command pattern är för att den som executar kommandot inte behöver veta vad kommandot är för något, dess information eller vad den gör allt är inkapslat i kommandot.

Lägg information som behövs för att utföra ett visst anrop i ett objekt som kan användas av klienter utan att de behöver veta vad anropet gör internt.

För att sätta det i ett exempel så om vi har klasser som airConditioner som har deras egna kommandon som Turn Termostat up, Turn Thermostat Down. Dessa kommandon kan då bli tilldelad till en knapp eller att den ska bli triggad om ett visst villkor uppfylls utan att behöva veta kommandots information.

2.3 Grundläggande objekt-orienterade koncept

Klass och objekt: En samling av olika typer eller/och till och med andra objekt.Objekt är ett element av en klass och har beteendet av sin klass.

Statisk och icke-statisk metod: Statiska metoder tillhör klassen och inte objektet. Icke-statisk metod tillhör ett objekt och kan därför inte nås från en statisk metod.

Variabel och attribut: Attribut (instansvariabler) är klassens variabler. Variabler är temporära objekt/typer som skapas i metoder.

Abstrakt klass, interface: Fungerar som en mall. Alla klasser som implementerar interfacet, eller ärver den abstrakta klassen, måste ha alla abstrakta metoder och attribut som mallen har (skapa sin egen version).

Statiska typer: Instansvariabelns deklarerade typ. Kan inte ändras. Kolla errors vid kompilerings tid.

Tex: Animal puppy = new dog();

Dynamiska typer: Ett objekt i heapen som kan ändras efter initiering. Kolla exekverings typ vid run time. Vid run-time kollar vi alltid på den dynamiska typen och inte den statiska, till exempel Animal klass är den statiska typen och Dog klassen är den dynamiska typen.

At running time då kollar den på puppy för att executa metoden eat(); och vid kompilerings tid kollar den på Animal.
Tex: method(puppy);

```
static void method(Animal var) {
        var.eat();
    }
```

Implicit argument – är det objektet som metoden, man ropar på, tillhör.
Tex: Tree björk = new Tree();
björk.grow();

Då är det implicita argumentet björk, med typen Tree.

Primitiv typ: Typer som redan finns i java-språket. Tex int, double, char, float...

Overriding: En subklass definierar sin egen version av en metod från sin superklass

Overloading: Objekt/klass har flera metoder med samma namn, men olika parametrar. Vilken som används bestäms vid kompilering (statiskt).

```
Tex     void run();
        void run(Boolean shoes);
```

Inkapsling: Ett sätt att sammankoppla variabler och metoder. Detta leder till att variablerna blir gömda för andra klasser och kan endast kommas åt via klassens metoder. (private, public, protected osv)

Konstruktor: Man anropar konstruktorn när man vill skapa en instans av det objektet

Referenstyp, alias: Referenstyp är typen av det objektet instansen pekar på. Alias betyder att två objekt/typer pekar på samma ”sak” i heapen.

Dynamisk bindning: Vilken metod som ska användas vid runtime. Om en klass har overridat en metod från sin superklass kommer den dynamiska bindningen att se till att den används vid runtime.

2.4 Avancerade språk mekanismer och tekniker

Immutability och Mutability: Objekt kan inte ändras efter initiering. Muterbara objekts tillstånd kan ändras men kan skapa problem när alias används. Pekarna pekar på samma objekt och på så sätt kan det bli problematiskt när ändringar ska göras.

Functional interfaces: En signatur (@FunctianalInterface) för ett interface som säger att det är ett funktionellt interface. Denna innehåller bara en metod och används vid lambda-uttryck.

Lambdas: En förkortad version av anonym klass.

```
MyFunktionType f = x -> x+5;
Behövs ett interface MyFuntionType. Krävs att det finns exakt en metod där.
@FunctionalInterface
int y = f.apply(5);
```

Design By Contract:

- **Förvillkor:** Predikat som ska gälla för att metoden ska få anropas. Användarens ansvar.
- **Eftervillkor:** Predikat som ska gälla efter genomfört metodanrop. Programmerarens ansvar

- **Invariant:** predikat som alltid ska gälla. Tex att square alltid ska ha lika långa sidor.

Exceptions: När man vet att ett fel kan inträffa, kan man kasta ett exception som isf visas vid run-time.

Refactoring: På hög nivå omstrukturera kod så att den blir mer översiktligt samtidigt som man behåller funktionalitet.

Defensive Copying: Istället för att skicka tillbaka en referens, skickar man tillbaka en kopia av objektet. Så vilka modifikationer du gör på kopian kommer inte påverka original objektet.

Ex: Konstruktor | instance to copy

```
Public Ninja(Ninja copy) {

    Ninja ninjaCopy = new Ninja();
    return ninjaCopy;
}
```

Method Cascading: Istället för att ”returnera” ändringar i objektet (void) returnerar man hela objektet efter ändringarna.

Tex:
```
this.height = height;
return this;
```

Defensiv programmering: Anta att det finns buggar i koden. Använda sig av tex exceptions för att kontra detta. Ha i åtanke att användaren alltid kan göra fel.

Mutate-by-copy: För att undvika att muterbara objekt ändras på fel sätt kan en ny kopia skapas som byter ut de värdena man vill ändra.

2.5 Arv, parameteriserade typer, code reuse, polymorfism

1. Vad är polymorfism? Vad är subtypspolymorfism? Vad är parametrisk polymorfism?

Polymorfism: är att flera subklasser under en superklass kan hanteras som om de vore instanser av superklassen. Det innebär att klasser med olika behov vad gäller implementeringen av en viss metod, ändå kan anropas på samma sätt. Den verkställande programkoden finns i respektive subklass, medan det gemensamma gränssnittet definieras i superklassen. Gemensamma metoder skall alltså finnas i superklassen.

Subtyps polymorfism: Ett objekt av en subklass kan användas som om det var ett objekt av sin superklass. Konkreta objekt beter sig på samma sätt (liskov).

Parametric polymorfism – Typ/metod kan parametriseras över en annan typ/metod. Definiera kod som är oberoende av en underliggande typ.
Tex: `class Vehicle<T>...`

Tex: Pair<Integer, String>

2. (V) På vilket sätt blir kod bättre av att man använder parametrisk resp. subtyps-polymorfism?

Man kan skapa en funktion som tar en lista av saker, och funktionen kan ändå funktionera utan att bry sig om vad de sakerna är för något. För att beskriva det mer i detalj så kan man exempelvis skapa en metod som returnerar antalet element i en kollektion. I listan kan du lägga in olika typer av element, och den kommer returnera ett svar. Fördelen med detta är att man inte behöver skriva om funktionen för varje typ av lista man lägger in.

3. Vad är kovarians? Vad är kontravarians?

Invarians:
Ex: List<Polygon> l = new ArraList<Polygon>();
När du vill både hämta och lägga till.
Kovarians:
Ex: List<? extends Polygon> l = new ArraList<Triangle>();
När du vill hämta något. Du är säker på att ? är en Polygon. Du kan inte lägga in något eftersom du inte vet vad ? är.

en metod som gör @Override kan t ex returnera en Square om motsvarande metod i superklassen returnerar Polygon

Kontravarians
Ex: List<? super Triangle> l = new ArraList<Polygon>();
När du vill lägga till något.

overriding ej möjligt, blir overloading.

4. Vad är delegering? Vad är arv?

Delegering: En klass kopierar funktionalitet hos original klassen för att få dess funktionalitet utan att ärva den.
Arv: När man vill "kopiera" något från parent klassen så använder man sig av arv, alltså ärver subklassen alla variabler och alla metoder från parent klassen.

5. Hur åstadkommer man kod återanvändning med delegering resp. arv?

Med delegering kan man återanvända funktionalitet hos original klassen för att lägga till extra funktionalitet utan att ändra något i parent klassen. Med arv kan man ärva från parent klassen för att visa förhållanden, till exempel dog ska kunna ärva från Animal klassen så den ska kunna identifiera sig som en animal. På detta sätt kan man utöka Animal klassen med fler förhållanden, i detta fall djurarter.

6. (V) När bör man använda arv? Varför säger vi att man ska föredra komposition framför arv (composition over inheritance)? Has-a, Is-A

Arv ger ett starkt beroende, vilket vi inte riktigt vill ha men ibland kan arv komma till nytta speciellt när vi inte vill duplicera kod som har ett Is-A förhållande.

Om tex en subklass overridar en metod från sin superklass som gör allt superklassen metod gör plus annat, subklassens metod kanske inte kompilerar när en ändring i superklassens metod görs. Det leder till att man måste ändra i både superklassen och subklassen. Användning av composition leder till att man bara behöver ändra i superklassen.

För att sätta det i ett exempel så om vi har klasser som Tree och Dog de båda behöver syre men Tree är ingen Animal och Dog är ingen Plant. Så därför vill vi använda komposition här för att vi bryr oss bara egentligen om vad dem gör och inte vad dem är för något. Därför är arv i detta fall dåligt.

7. Vad är en typkonstruktor? Vad är en typparameter?

Typ Konstruktor: Arrays: [] kan ses som en typkonstruktor för array-typer. Vi har inte en faktisk typ förrän vi anger typen för elementen, e.g. String[].
Generic types: E.g. ArrayList<_> är en typkonstruktor. Vi måste ange en argument-typ för att vi ska ha en faktisk typ, som vi kan skapa objekt av, e.g. ArrayList<String>.
I haskell är [] och Maybe typkonstruktorer.

Typ Parameter: Public class Pair<A,B> A och B är en typ parameter

8. Vad är en typvariabel? Vad är ett wildcard?

En typ-parameter (T) är en variabel som representerar en typ. Vi vet inte vilken typ detta är – det bestäms av det argument som ges vid instansiering, och kan vara olika typer olika gånger vi instansierar.

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

Unbounded Wildcard: use ?, List<?> this is a list of unknown type.

Upperbound Wildcard: if you want to write a method that works on List<Integer> or double etc you can achieve this by using an upperbounded wildcard. To use it use the ? followed by extends. Like in the example: List<? extends Number>

Lowerbound Wildcard: a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type. A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: <? super A>

9. I vilken utsträckning tillåter Java varians vid metod-overriding?

kovarians i returtypen för att man kan returnera subtyper till klassen eller klassens typ.
Invarians i arguemnttyp för att annars så skulle det vara overloading.

10. Vad innebär nyckelorden extends och super?

Super: Om ens metod overridar en av superklassens metoder kan man använda sig av Super för att definiera sin egna metod.

Extends: använder vi när vi vill ärva alla variabler och metoder hos ens parent class med ett undantag för privata variabler och metoder.

11. (V) När bör man använda extends och super? När man vill ta och hämta något. Get-put principen.