# Objektorienterad programmering

Tentaförberedelse

### Objektorienterad modellering och OOP i allmänhet

**Vilka är några önskade egenskaper hos mjukvarusystem?**

Hög kohesion - varje klass ska representera en och endast en roll

Låg koppling - beroendet mellan olika klasser ska vara minimalt

Inkapsling - det som inte ska synas utåt är inkapslat och endast åtkomligt via publika metoder. Detta ger kontroll över hur data ändras i programmet (kontrakt) och definierar ett publikt gränssnitt (hur man manipulerar objektet).

**Vad kännetecknar objektorienterad programmering och vilka är fördelarna?**

Objektorienterad programmering innebär att en varierande uppsättning av objekt interagerar med varandra. Varje objekt innehåller (äger) data (fält, såsom attribut) och kod (procedurer, såsom metoder). Då objekten äger datan och metoder inuti kan den komma åt och ändra sin egen data (i java via ‘this’) och kan interagera med andra objekt via accessmetoder.

Fördelarna med objektorienterad programmering är:

* Modularisering   
  Varje objekt (eller grupp av objekt) kan ses som en modul som underlättar bland annat felsökning men strukturen av ett program när det växer.
* Återanvändning (via arv)  
  Många klasser kan dela funktionalitet, och istället för att skriva om detta för varje klass kan alla klasser ärva från en gemensam basklass (superklass).
* Flexibilitet (genom polymorfism)  
  En variabel har inte en fix datatyp utan kan “ändras” under programmets gång via dynamisk bindning. Detta görs genom att t.ex. en funktion tar in en superklass som parameter. Sedan anropas funktionen med en subklass. Då kommer parametern vara en referens till subklass-objektet, men endast komma åt funktionaliteten i superklassen.

#### Nyckelbegrepp:

* Modellering av ”real world”  
  Innebär att objekten i programmet modellerar objekt i verkligheten. Såsom att man kan ha en klass *Car* eller *Dog*.
* Modularisering  
  Innebär att varje objekt (eller grupp av objekt) ses som en modul av hela programmet. På detta sätt delas programmet upp i mindre delar och problemet kan enklare lösas. T.ex. skriva ett program som simulerar bilkörning, då kan programmet modulariseras till *motor*, *hjul*, *säten* osv. Dessa moduler bygger upp hela bilen men har inget att göra med varandra (låg koppling).
* Dataabstraktion  
  Innebär att dölja komplexitet och onödiga detaljer för användaren. “Användaren” är då en annan programmerare som användare din klass. Detta gör att användaren lätt kan implementera annan logik ovanpå din klass utan att fullt förstå din implementation.   
  T.ex. En klass för en kaffemaskin skrivs *CoffeeMachine* men alla logik för att beräkna om tillräckligt med kaffebönor finns, tillräckligt med vatten finns osv. döljs och endast funktioner som *brewCoffee()* blir publika.
* Inkapsling  
  Innebär att dölja direkt åtkomst till intern data för den som användaren klassen och istället ger åtkomst till vald data genom åtkomstmetoder (“getter” och “setter”). Inkapsling innebär alltså att dölja implementation bakom ett API. Poängen med detta är att klassen då kan ändras internt (implementationen kan ändras) utan att påverka den som ändras klassen. T.ex. klassen *Window* modellerar ett fönster på datorn och *getWidth()* är en åtkomstmetod. Programmeraren vill nu veta hur många gånger denna kallas. På grund av att inte *Window.width* inte är publik behöver endast en rad i *getWidth()* läggas till för en räknare, utan att påverka den som kallas på funktionen.
* Återanvändning  
  Innebär att olika klasser som är olika typer av samma sak kan ärva från superklasser som beskriver den gemensamma funktionaliteten. T.ex. en limousin och racer-bil är båda typer av bilar, som skulle kunna implementeras att ärva från en superklass *Car* och därmed återanvänds koden i *Car*.
* Hög kohesion  
  Innebär att varje komponent samverkar för att lösa modulens ansvarsområde, utan att behöva samverka med komponenter i andra moduler - hur väl modulen klarar ett uppdrag på egen hand och hur väldefinierat dess ansvarsområde är.
* Låg koppling  
  Innebär att modulers beroende av varandra är lågt. Genom att eftersträva lågt beroende blir modulerna flexiblare.
* Arv  
  Innebär att klasser kan ärva från andra klasser för att slippa upprepa kod och minska risken för att missa implementera funktion som krävs.

#### Övergripande begrepp

* Paket  
  Ett sätt att gruppera relaterade klasser, som en mapp grupperar filer. Ett paket kan innehålla andra paket. Klasser i samma paket kan komma åt fler variabler i varandras klasser.
* Klass  
  En användardefinierad ritning eller prototyp som objekt kan skapas från
* Objekt  
  En instans av en klass med egna värden av klassens definierade data
* Datamedlem (dataattribut)  
  Fält i en klass som är menade att användas för att modellera klassen. T.ex. *Window*-klass kan ha *width* och *height* som datamedlemmar
* Metod (medlemsfunktion/operation)  
  Funktioner kopplade till en klass som är menade att agera på eller med hjälp av datamedlemmarna i en klass. T.ex. *window.clear()* kan rensa vad som ritats i just den instanser av klassen *Window*.
* Konstruktor  
  En speciellt metod som finns i varje klass och används för att initiera alla värden (datamedlemmar) i ett objekt av klassen. Konstruktorn finns även om användaren inte har definierat den själv och sätter då default-värden på medlemmarna.
* Accessmetod  
  En funktion som används för att hämta en privat datamedlem i en klass. Accessmetod kallas ofta för getter.
* Mutator-metod  
  En funktion som används för att ändra en privat datamedlem i en klass.  
  Mutatormetod kallas ofta för setter.

#### Saker att kunna:

* Tillgänglighet/synlighet. Viktigt: Vad är poängen?   
  Genom att begränsa vad användaren av en klass kan se och göra med medlemmarna (och skapa ett statiskt API) kan framtida ändringar göras i implementation utan att ändra användningen av klassen.
* Vad menas med ett interface/gränssnitt för en klass?  
  Ett interface för en klass beskriver hur en klass är tänkt att användas. Ett interface är en uppsättning av funktioner som måste definieras (implementeras) av den klass som implementerar interfacet. På detta sätt kan det säkerställas att den klass innehåller alla funktioner som krävs för en uppgift, och polyformism kan användas.

##### Relationer mellan objekt:

* + Beroende/dependency  
    Syftar på vilken annan kod en klass måste känna till för att kunna fungera.   
    T.ex. en klass som modellerar en lärare måste känna till klassen *Person* då en lärare är en person, men klassen person måste inte känna till lärare.
  + Association  
    Syftar på typ av relation mellan två klasser som beskriver en aktivitet som utförs mellan dem. T.ex. en student går på en kurs, och en lärare lär ut en kurs. Då har både student och lärare en association till klassen *Kurs*.
  + Aggregat  
    Syftar på en relation mellan två klasser där den ena klassen består av den andra klassen men på ett sätt där de kan finnas utan varandra.  
    T.ex. ett klassrum innehåller elever, men klassrum kan finnas utan elever inuti det.
  + Komposition (vem är du)  
    Syftar på en relation mellan två klasser där den ena klassen består av den andra klassen men på ett sätt där de **inte** kan finnas utan varandra.  
    T.ex. ett hus innehåller rum, men ett hus kan inte existera utan rum i det (finns alltid minst ett rum).

##### Relation mellan klasser:

* + Arv  
    Innebär att basera funktionaliteten i en klass på en redan existerande klass. Detta är en av fördelarna med OOP då det bidrar till *återanvändning*. Arv kan även ske på abstrakta klasser som är en klass som har minst en abstrakt metod. Abstrakta metoder är metoder som inte ännu har en definition och måste implementeras av den som ärver klassen.
  + Interface  
    En speciellt typ av arv där endast “klassen” som ärvs från endast innehåller abstrakta metoder. Abstrakta metoder är metoder som inte ännu har en definition och måste implementeras av den som implementerar interfacet.

**Kunna ange lämplig relation i olika fall. Kunna göra välmotiverat val mellan arv resp. composite aggregation i situationer där båda vägarna verkar möjliga.**

### Analys och design, modellering med UML, designmönster

Vad syftar analys-, design- resp. implementationsfasen till?

* Analysfasen  
  Syftar på stadiet i utvecklingen där data samlas in om kring problemet för att förstå vad problemet gäller och vad som krävs för att lösa problemet.
* Designfasen  
  Syftar på stadiet i utvecklingen där programmet struktureras upp översiktligt. Vilka komponenter utgörs systemet av och hur är det översiktligt tänkt att prata med varandra? Designfasen baseras på analysfasens resultat.
* Implementationsfasen  
  Syftar på stadiet i utveckling där programmet praktiskt konstrueras enligt designen från designfasen.

Vad är UML och vad används det till?   
UML står för *Unified Modelling Language* och är till för att visualisera ett objektorienterat programs struktur. Det visualiserar klasser, deras datamedlemmar och metoder, och hur de pratar med andra klass (vilken relation de har till varandra). UML är inte bunden till något specifikt programmeringsspråk och kan modellera alla typ av system, det används till och med till olika affärsprocesser helt utanför programmering.

Vad är:

* Användningsfall (use cases)  
  Syftar på vilken funktion olika *aktörer* är tänkt att använda ens program. Aktörer syftar på det olika typer av användare.
* Scenarios  
  Syftar på en kort beskrivning på ett sätt en användare är tänkt att använda ens program. T.ex. användningsfall för ett online-mataffär: Användare loggar in i “Mina Sidor” osv.
* Klassdiagram  
  Syftar på ett diagram uppbyggt via UML som visualiserar ens programs struktur. Diagrammet visar vilka klasser som finns och vilken relation de har till varandra - aggregat, komposition, association.
* Sekvensdiagram  
  Syftar på ett diagram som visar vad i ens program som händer i vilken ordning - “När en användare förfrågan kommer skickas den först dit, sedan efter svar skickas förfrågan vidare dit, sedan får användaren tillbaka ett svar”.

#### Saker att kunna

1. Kunna välja relation i specifikt fall.
2. Syntax, hur ser den grafiska notationen ut?
3. Du ska utifrån (enkla) problembeskrivningar kunna ta fram rimliga användningsfall, med scenarios, och rita motsvarande klass- och sekvensdiagram.

#### Designmönster

Vad är ett designmönster?  
Rent generellt är designmönster en namngiven lösning på ett, i en viss omgivning, ofta återkommande problem. I systemutveckling handlar designmönster på vilket sätt man strukturerar upp ett program för att lösa ett problem.

#### Saker att kunna

Beskriva mönster och när det kan användas: Singleton, Iterator, Composite, Proxy, Strategy, SubjectObserver, Facade, Model-View-Controller

* Singleton  
  Ett designmönster där en klass modelleras på sådant sätt att endast ett objekt kan finnas. Detta är ett bra designmönster vid t.ex. renderare i en grafikapplikation, då det endast krävs en renderare för att få en bild på skärmen, och det finns ingen anledning att ha fler än en.
* Iterator  
  Ett designmönster som går ut på att löpa igenom objekten i en behållare som behållare själv bestämmer.  
  Java använder detta för sin “for-each” loop över LinkedList och ArrayList.
* Composite  
  Ett designmönster som går ut på att behandla ett grupp av klasser som en singleton instans, dvs. en enhet. Ex. ett filsystem som börjar med en *Directory* som innehåller nodes (grenar eller löv i trädet). En directory kan innehålla directories som i sin tur också innehåller directories. Hela trädet av filer och mappar kan då ses som en singleton enligt composite-designmönstret.
* Proxy  
  Ett designmönster där en klass agerar som mellanhand mellan två andra klasser, alltså som ett interface till andra klasser. För att klass A ska kunna prata med klass B måste den göra det via proxyn mellan dem. Detta mönster används när resursen bakom proxyn är tung resurs (såsom stor objekt i minnet) eller annat objekt som är omöjligt att duplicera. Därmed kan *proxy* ses som en wrapper som begränsar åtkomst.
* Strategy  
  Ett designmönster som möjliggör val av algoritm vid runtime, istället för att implementera en enskild algoritm direkt (compile-time).   
  I praktiken gör detta att ett t.ex. en funktion som löser ett problem tar in ett objekt som parameter som bestämmer vilken algoritm som ska användas. I Java görs detta via override, men andra språk kan skicka t.ex. funktionspekare.
* Observer  
  Ett designmönster som går ut på att ett objekt (Subject) håller i en lista av *dependents* (Observers), och meddelar detta när ett stadie har ändrats. I praktiken är detta är väl passande designmönster för händelsehanteringssystem (event handling), då en klass kan “registrerar sig” som observer av ett specifikt event hos subject. När det eventet inträffar blir denna klass meddelad (ofta i form av ett funktionsanrop).
* Facade  
  Ett designmönster som går ut på att skapa en sort fasad framför en mer komplex underliggande struktur. Detta används för att förbättra läsbarheten, förenkla att bygga på strukturen och skapa ett mer kontextspecifikt gränssnitt till underliggande generellare gränssnitt. En fasad är alltså som ett extra lager gränssnitt, fast mer det är mer relevant i kontexten.
* MVC  
  Ett designmönster som går ut på att separera data (Model) och presentation (View) så att förändringar i presentationslagret i påverkar datorhanteringen, samt att data ska kunna omorganiseras utan att behöva ändra något i presentationslagret. De separeras med hjälp av en mellanliggande komponent, *Controllern*. Controllern använder data i *Model* för att skriva ut på *View*.

### Allmänt om Java

* Vad är plattformsoberoende och hur kan Java vara det?  
  När Java-kod kompileras blir inte detta maskinkod, utan Java-kod går via en Java-kompilator till att bli Java Bytecode. Detta är kod som sedan kan avläsas av en interpretator som finns på olika operativsystem. Java har alltså ett extra egenskapat abstraktionslager som gör att koden blir plattformsoberoende.
* Interpretering versus kompilering till maskinkod - fördelar och nackdelar?  
  Fördelen med interpretering är att programmet blir enklare att implementera på olika plattformar och därmed ett bekvämare språk.  
  Fördelen med kompilerad kod mot interpretering är att den har bättre prestanda och att många optimeringar kan utföras vid kompilering.

#### Vad är:

* JavaVirtualMachine/javatolken/interpretatorn  
  Interpretatorn har ansvar att tolka Java ByteCode till faktiska CPU-instruktioner, alltså en form av översättare från Java Bytecode till exekverbar kod.
* Javabytekod  
  Kod i form av en typ av lista med instruktioner som interpretatorn måste översätta till exekverbar kod.
* Livstid för primitivt data och referensvariabler?   
  Livstiden för primitiv data och referensvariabler är i dess scope. Utanför en klass innebär detta när while-loopen, if-satsen, for-loopen, funktionen tar slut (taget att variabeln blev deklarerad där). Primitiv data och referensvariabler i klasser lever så länge objektet lever. Primitiv data och referensvariabler kopieras alltid och har därför oftast livstiden inom det lokala scopet.
* Objekts livstid?   
  Ett objekts livstids skiljer sig då den inte kopieras när den t.ex. skickas som parameter till funktioner, utan en referens till den skickas. Ett objekts livstid är därför slut när det inte finns något referens kvar till objektet.
* Garbage Collection  
  Syftar på Javas sätt att rensa upp minne som inte längre används. T.ex. när det inte finns någon referens till objekt längre.
* Primitiva datatyper för heltal, flyttal, tecken.  
  Syftar på de inbyggda typer i språket Java, dvs. utan några importerade standardbibliotek - int, float, double osv.
* Vad är skillnaden på primitivt data och data av klasstyp?   
  Syftar på de inbyggda typer i språket Java, dvs. utan några importerade standardbibliotek - int, float, double osv.  
  Klasstyp syftar på användarskapad datatyp beroende på vad som ska modelleras.   
  Klasstyper kan även ärva från andra klasstyper, det kan inte primitiva datatyper.
* Vad är en referensvariabel?   
  En typ av variabel som refererar (“hänvisas”) till ett objekt av en klass. När nyckelordet *new* används för att skapa ett objekt av en klass, får man returnerat en referens till objektet. I Java arbetar man alltså tekniskt sett aldrig direkt på objektet, utan via en referens. På detta sätt kan Javas garbage collection hålla kolla på när ett objekt inte längre används. Referensvariabler finns endast för objekt, inte för primitiva datatyper.
* Vad är ett alias?  
  Ett alias kan ses som en referens till samma minna som en annan variabel. Om *b* deklareras med *B[] b = new B[10]* och *A[] a = b*, kommer *a* inte kopiera *b* utan peka på samma minne som *b*, som en referens. Detta gör att alla ändringar på a kommer påverka b, och vice versa.

#### Begrepp

* Vad innebär jämförelsen o1 == o2?   
  Referenser jämförs och ser om det “pekar” på samma objekt. Alltså ej “värdet” på objektet.
* Hur bör vi göra för att jämföra innehållet i 2 objekt?   
  o1.equals(o2);
* Referensen null  
  Syftar på en referens som inte pekar på något objekt.
* Paket/package   
  Syftar på en typ av grupp som klasser delas in i
* Hur placerar man en klass i ett paket?   
  Package <package namn>
* Poänger med paket?   
  Paket i Java används för att minska risken för namnkollisioner, kontrollera tillgång till funktioner (genom package private), bättre filstruktur (paket är en mapp på disken), enklare användning av klasser, interface och enums.
* Vad är API?   
  Application Programming Interface är som en fasad för ett program som definierar vilka externa interaktioner som får ske med programmet. Till Java finns APIer såsom JavaFX som kan anropas för att skapa fönster och GUI.

#### Arrayer

* Vad menas med att ett objekt är immutable/oföränderligt? Poäng?  
  Oföränderliga objekt syftar på ett objekt som antar ett värde vid sin konstruktion och inte ändras därefter. Fördelen med ett sådant objekt är att den garanterar samma funktionalitet genom dess livstids, samt att den i säkerhet kan användas av flera trådar samtidigt (inga race conditions kan uppstå om data inte förändras).
* Elementära operationer på String-objekt.
* Hur skapar man en array för t ex heltal?   
  *int[] heltal = new int[<storlek>];*
* Hur vet man maximala storleken?   
  *heltal.length*
* Hur skapar man en array för objekt?   
  *MyClass[] allObjects = new MyClass[<antal>];*
* Vad innehåller arrayen?   
  I början endast *null* referenser, men sedan referenser till objekt.
* Hur lägger man till objekt?  
  *allObjects[<index>] = new MyClass();*

#### Klasser

All kod skrivs i klasser (Java) Datamedlemmar och metoder.

* Vilka nivåer finns för tillgänglighet/synlighet?   
  Public, (“package”), protected och private
* Vad gäller om man inte anger tillgänglighet?  
  Package private, dvs. endast de i samma paket har tillgång.
* Skillnad mellan datamedlemmar och lokala variabler?   
  Datamedlemmar är variabler som finns inuti en klass, lokala variabler finns i någon form av scope.
* Vad innebär static för datamedlem respektive metod?   
  Static datamedlem syftar på en datamedlem som är samma för alla objekt av klassen. Om ett objekt ändrar den, ändras den för alla i klassen.  
  Static metoder syftar på metoder som är samma för alla objekt. Det som är speciellt är att dessa metoder inte får ändra icke-statiska datamedlemmar, då de inte är bundna till något objekt.
* Vad innebär final?   
  Variabeln kan inte ändras efter den har blivit initierad. Variabeln kan dock blir initierad med hjälp av konstruktorn om den inte redan fått ett värde (genom in-line initiering).
* Parameteröverföring  
  I Java är alla parameter alltid pass-by-value. Dock vid när det gäller objekt, är det referensen som är överförd i kopia, och objektet kopieras inte.

### Felhantering med exceptions (undantag)

* Vad är exceptions och varför används dem?   
  Exceptions används för att signalera att programmet är i ett stadie som inte är idealt. T.ex. Vid inläsning av fil från disk kan något orsaka att den blev avbruten mitt i läsningen och kunde inte läsa filen fullt ut.
* Hur implementeras enhetlig felhantering (hantera fel på lämplig nivå)?  
  Exceptions kräver: try-catch-finally struktur där try ramar in i vilken kod som det riskerar att uppkomma exceptions i, catch hanterar undantaget beroende på vilket typ och finally är kodsnutt som alltid körs i slutet.  
  Dock behövs endast endast en av catch och finally.
* Vad innebär throw - hur kan exceptions genereras?   
  *throw* skapar ett nytt exception objekt och argument skickas in (t.ex. en sträng om vad som hände eller annan data för att åter få ett idealt stadie).
* Vad innebär try?  
  Ramar in koden som riskerar att kasta exception
* Vad innebär catch?  
  Hanterar exception beroende på dess typ
* Vad innebär finally?  
  Kodsnutt som alltid körs i slutet, oavsett om ett exception kastades eller inte
* Vad är checked resp. unchecked exceptions?  
  Checked exceptions syftar på exceptions som går att förutspå kan kastas i kompileringen. T.ex. Jag skriver en IO-klass som läser filer, men om det gick dåligt kastar jag ett exception. Funktionen måste då markeras med “throws <exception-typ>”.  
  Ett unchecked exception syftar på aritmetiska exception (divide-by-zero), “Index out of bounds” osv.
* Hur kan man skilja olika exceptions åt i catch-delen?  
  *catch(<mest specifika>)  
  {  
  }  
  catch(<mindre specifikt>)  
  {  
  }  
  catch(Exception e) // Alla resterande exceptions  
  {  
  }*
* Vad händer om en exception inte fångas i den metod den genereras?   
  Det “klättrar” uppåt i stack-frames tills att det antingen når ett try-block eller utanför main-funktionen (i vilket fall programmet kraschar).
* Hur kan icke-fångade exception användas praktiskt?  
  En funktion som *tryDoThis()* kan kasta exception om något inte gick att utföra, och “returnerar” information varför det gick fel.
* Vad är “Stack unwinding”?  
  Det minne som fanns i stack-framen för funktionen försvinner när exception kastas och inte fångas. Det vill säga, den process som gör att det inte sker minnesläckor vid användning av exceptions (samma som sker vid return).

### Generics

* Vad är generics?  
  Syftar på parametriserade datatyper och innebär att låta klasser, metoder och interface vara uppbyggda på olika datatyper. Generic Entity kallas en klass, metod eller interface som är parametriserad.
* Vad är poängen med generics?  
  Genom att låta klasser vars syfte är att operera på en datatyp vara parametriserade minskar mar upprepning av kod. T.ex. en tvådimensionell matematisk vektor utförs operationer på två variabler, men typen av dessa spelar ingen roll så länge det är ett form av tal. Då kan man parametrisera denna så att den fungerar på int, float och double osv.
* Hur skrivs en klass med en typparameter?  
  *class Test<T>  
  {  
  }  
    
  Test<Integer> myIntTest = new Test<Integer>(15);  
  Test<Double> myDoubleTest = new Test<Double>(15);*

#### Arv

* Vad ärvs?  
  Alla metoder och datamedlemmar som inte är markerade som private.
* Hur fungerar tillgängligheten protected för subklass och paket?  
  Protected begränsar metoden eller datamedlemmen till att endast finnas för sig själv och subklasser, samt alla andra klasser som ingår i samma paket.
* Hur fungerar nyckelordet super (i konstruktor resp. i annan metod)?  
  *super()* används endast i konstruktorn för att kallas på superklassens konstruktor  
  *super.dataMedlem* / *super.metod()* används för att komma åt superklassens datamedlemmar eller metoder
* Vad är en subtyp?  
  Klassen som ärver från en annan klass (superklass/supertyp)
* Vad menas med ”down cast”?   
  Behandla ett superklass-objekt som om det vore dess subtyp. En supertyp kan inte bestämmas vad det faktiskt är vid compile-time, och om down-casten inte visar sig vara korrekt kommer ett exception kastas.   
  Ex.   
  *class A {....*  
  *class B extends A {....  
  class C extends A {...  
    
  A myA1 = new B(); // Dynamisk binding  
  A myA2 = new C();   
  B myB1 = (A)myA1; // Kompilerar då det finns en chans att myB faktiskt är class A*  
  *B myB2 = (A)myA2; // Kompilerar men kommer kasta exception då A != C*
* Vad är dynamisk bindning?  
  När ett objekt av en supertyp skapas med ett värde av en subtyp.  
  Ex.   
  *class A {....*  
  *class B extends A {....  
    
  A myA1 = new B(); // Dynamisk bindning*
* Vilka regler finns vid implementering av interface resp. rent arv (bl.a. multipelt arv?).  
  En klass kan endast ärva från en klass, men kan implementera flera interface.
* Hur används nyckelordet instanceof – hur kan det användas?  
  Returnerar sant eller falskt om ett objekt är en instans av en viss klass.   
  Fungerar även för superklasser.  
  Ex.   
  *class A {....*  
  *class B extends A {....  
    
  A myA = new B(); // Dynamisk bindning*  
  *if (myA instanceof B) -> TRUE  
  if (myA instanceof A) -> TRUE*
* Vad är basklassen Object för något? Hur kan denna klass användas?  
  Den mest elementära klassen i Java som alla klasser ärver från. Den kan används när en funktion tar vilket objekt som helst som en parameter.

#### Att kunna

* Regler vilka typer av objekt en referens får referera till – undersöks ”runtime”.
* Regler för metodanrop från en referensvariabel – undersöks ”compile time”.

### Strömmar

Vad är en ström? Input och output  
En sekvens av data som antingen används för att skriva eller läsa. När en ström skapas innebär det mer eller mindre att öppna en väg för data att flöda. Inputstream avser en ström som läses från och en output stream avser en ström som skrivs till. I praktiken är detta t.ex. att läsa från eller skriva till fil.  
En ström kan används till mer än bara filer, utan det går att öppna strömmar för att läsa från minne som används av programmet, t.ex. en stor array.

### Grafiska användargränssnitt och händelsehantering

#### Typer av grafiska komponenter:

* Vad är ett fönster?  
  Fönster som en grafisk komponent syftar på ett fönster på skärmen som kan visa olika typer av grafiska komponenter och uppdateras vanligen varje gång något nytt ska visas på skärmen. Ett fönster hanterar också ofta event såsom musklick och tangentbordsklick.
* Vad är en behållare (parent/pane)?  
  En behållare, t.ex. en pane/group/parent, håller i grafiska komponenter, som kan visas i fönstret. En behållare kan hålla i flera behållare.
* Vilka övriga komponenter finns i ett grafisk gränssnitt?  
  UI-controls: radioknapp, checkbox, textinput osv.  
  Visuellt: Bilder, former

### Trådar

* Hur skapas och hur avslutas en tråd i Java?   
  En tråd skapas genom att ärva från Javas inbyggda Thread och overridea metoden *run*. Det går också att använda s.k. lambda-uttryck för att skapa en tråd. En tråd avslutas så fort den är klar med funktionen *run()*.
* Hur används klassen Thread resp. interfacet Runnable?   
  En runnable kan användas för att skapa en thread (dvs. parameter i konstruktorn) eller via funktion som *runLater()*.
* Hur kan man lösa Race Condition-problem i Java?  
  Genom att använda mutex eller funktionen *runLater()*.
* Vad menas med monitor (lås)?   
  En monitor är en entitet som äger ett lås, vilket är varje objekt i Java (från klassen Object). Detta gör att nyckelord som *synchronized(<objekt>){....* låser de inbyggda låset i objektet.
* Vad är synkroniserade metoder?  
  Synkroniserade metoder innebär att endast ett objekt av klassen i taget kan använda metoden.