( = svår)

1.
Skriv en metod **isEven()** som kontrollerar om ett givet tal är jämnt eller inte. Den ska returnera en **boolean**. Skriv sedan ett testprogram som består av att ni anropar metoden ett par gånger med olika indata, för att se till så att den funkar ordentligt.
Tips: ert testprogram kan ni helt enkelt skriva i main-metoden.
Tips: man kan kolla om ett tal är jämnt med *modulus,* dvs kolla vilken rest vi får när vi dividerar talet med 2. Uttrycket blir i%2==0.

2.
Skriv ett program som frågar användaren efter ett tal som ligger i intervallet 0<=x<10 (dvs talet måste vara större än eller lika med 0 och mindre än 10).
Om talet ligger i intervallet skriv ut lämplig gratulationsfras och avsluta programmet, annars fråga användaren igen tills hen lyckas.

3.
Skriv ett program som kontinuerligt hämtar en textsträng från användaren och skriver ut den på skärmen. Programmet ska fråga om och om igen, fram tills dess att användaren skriver in samma sak två gånger i rad. Då ska programmet avslutas.

4.
Skriv ett program som frågar användaren efter två tal och som sedan beräknar medelvärdet av dem. Dela upp programmet i två metoder: en som hämtar en **integer** från användaren, och en som beräknar medelvärdet av två tal. Anropa dessa metoder på lämpliga sätt.

5.
Skriv en metod **isAuthorised()** som frågar användaren efter tre lösenord och som kontrollerar om alla tre är korrekta. Returnerar **boolean**. Ordningen användaren matar in lösenorden i ska inte spela någon roll - om lösenorden är “piggy”, “snuff” och “bark” så ska ordningen “snuff”,”bark”,”piggy” också accepteras.

Använd **isAuthorised()**-metoden i er main-metod och skriv ut en lämplig gratulationsfras ifall användaren lyckats mata in alla lösenorden korrekt.

Om användaren inte lyckades ska programmet låta henne försöka fyra gånger till, så att antalet totala inloggningsförsök man har på sig är fem. Om användaren svarar fel på det femte försöket, ska programmet avslutas. Om användaren däremot svarade rätt på det femte försöket, ska ett specialanpassat meddelande visas för att lugna användaren.

6.
Skriv en metod **anyIsTrue()** som tar fyra argument, och som kollar om något av de fyra sanningsvärdena är sant. Returnerar en **boolean**. Skriv ett testprogram.

7.

Utöka ert program från förra uppgiften med ytterligare en metod, **allAreFalse()**, som returnerar sant om alla dess fyra argument är falska (den fungerar alltså precis tvärtom som **anyIsTrue()**. Utöka också testprogrammet så att det testar den nya metoden.

Tips: Ni kan återanvända **anyIsTrue()** när ni svarar på denna uppgift, så att ni inte behöver skriva ytterligare ett logiskt uttryck...

8.
Skriv en metod som tar genomsnittet av alla tal i en **array** och returnerar det. Skriv ett testprogram.

Tips: metodens argument ska vara en **array** av **doubles** och den ska returnera en **double**.

9. 
Nu ska ni skriva ett program som låter användaren mata in strängar. Programmet ska fortsätta fråga efter nya strängar så länge strängarna blir längre och längre, dvs så länge varje sträng har fler tecken i sig än den föregående. Om en sträng med lika många eller färre tecken matas in ska programmet sluta fråga efter strängar, skriva ut alla som hittills har matats in, och till slut avslutas.

Tips: eftersom ni inte vet hur många strängar som matas in, måste ni på något sätt förändra arrayens storlek efter varje inmatning. Man kan dock inte förändra storleken på en redan skapad array - hur ska ni då gå tillväga?

10. 
Skapa en metod som tar emot en **array** av **integers**, och som sedan *sorterar* den enligt sorteringsalgoritmen *bubble sort*, och returnerar den. Skriv ett testprogram som testar metoden med en osorterad array - skriv då ut arrayen både innan och efter sorteringen.

Exempel på osorterad lista: 157202640
Efter att vi sorterat den: 001224567

Beskrivning av algoritmen:
Bubbelsortering (engelska *Bubble sort*) är en enkel sorteringsalgoritm men inte särskilt effektiv. Metoden går ut på att man upprepade gånger går igenom det område i listan som ska sorteras och gör parvis jämförelser av intilliggande element.
När två intilliggande element ligger i fel ordning byter man plats på dem. Varje gång man gått igenom ett område kommer det sista talet att ha hamnat på rätt plats. Nästa gång reducerar man därför det område man går igenom med ett.

Efter hand som man gör sorteringen kommer listan i botten bli alltmer korrekt och de överblivna talen "bubblar" uppåt, därav namnet på sorteringsalgoritmen.

Bra bild på hur algoritmen funkar: http://upload.wikimedia.org/wikipedia/commons/c/c8/Bubble-sort-example-300px.gif

Varning:
Det finns pseudo-kod för hur man kan programmera bubble sort på dess wikipedia-sida. Om ni kollar på den blir uppgiften avsevärt lättare! (Försök gärna tänka ut det själva först)

11. 
Uökning av förra uppgiften.
Skriv nu en metod som tar emot en **array** av **heltal**, sorterar den med hjälp av *Comb sort*, och returnerar den sorterade arrayen.
Här är en beskrivning av algoritmen (kopierat från Wikipedia):

The basic idea is to eliminate *turtles*, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. *Rabbits*, large values around the beginning of the list, do not pose a problem in bubble sort.

In bubble sort, when any two elements are compared, they always have a *gap* (distance from each other) of 1. The basic idea of comb sort is that the gap can be much more than 1 (Shell sort is also based on this idea, but it is a modification ofinsertion sort rather than bubble sort).

In other words, the inner loop of bubble sort, which does the actual *swap*, is modified such that gap between swapped elements goes down (for each iteration of outer loop) in steps of shrink factor. i.e. [input size / shrink factor, input size / shrink factor^2, input size / shrink factor^3, ..., 1]. Unlike in bubble sort, where the gap is constant i.e. 1.

The gap starts out as the length of the list being sorted divided by the *shrink factor* (generally 1.3; see below), and the list is sorted with that value (rounded down to an integer if needed) as the gap. Then the gap is divided by the shrink factor again, the list is sorted with this new gap, and the process repeats until the gap is 1. At this point, comb sort continues using a gap of 1 until the list is fully sorted. The final stage of the sort is thus equivalent to a bubble sort, but by this time most turtles have been dealt with, so a bubble sort will be efficient.

Här är även en animation som visar hur algoritmen fungerar:

http://upload.wikimedia.org/wikipedia/commons/4/46/Comb_sort_demo.gif

Varning:
Även här finns det pseudo-kod för hur algoritmen är programmerad på dess wikipedia-sida, så vill du lösa den utan hjälp bör du hålla dig ifrån den!