En nackdel med fält är att det är besvärligt att lägga till och ta bort data om dessa inte ligger sist i fältet. För att placera en ny cirkel först i ett fält måste övriga data i fältet flyttas ett steg innan den nya informationen kan infogas. På motsvarande sätt måste data flyttas då ett värde tas bort från ett fält för att slippa ett "tomrum". För att utföra dessa operationer måste programmeraren skriva mer kod, förutom att förflyttningarna tar tid vid programkörningen. En annan nackdel med fält är att dess storlek måste bestämmas då det skapas. Ofta vet vi inte hur stort fält vi behöver för att lagra data och av den anledningen skapas kanske ett onödigt stort fält och programmet behöver då mer minne för att kunna köras.

14 Samlingar

Fält används för att lagra data som logiskt hör ihop. Det är till exempel enklare att skriva ett program som hanterar flera cirklar om cirklarna lagras i ett fält i stället för i ett antal cirkelvariabler. Fält har dock en del begränsningar och av den orsaken har Java kompletterats med andra lagringsmöjligheter av data. I det här kapitlet ska vi undersöka tre vanliga och användbara klasser som kallas *ArrayList*, *Stack* och *PriorityQueue*. Dessa kan ses som komplement till fält och har anpassats för att underlätta hanteringen av vanliga programmeringsuppgifter.

14.1 ArrayList

Ett objekt av klassen ArrayList kan betraktas som ett fält med ett tillägg av ett stort förråd av färdiga metoder vars syfte är att förenkla arbetet för programmeraren. Datat lagras på samma sätt som i ett fält, det vill säga ett efter ett i en direkt följd i minnet. I tabellen nedan visas några av de metoder som hör till klassen ArrayList. Flertalet av dessa kommer att förklaras utförligare i detta avsnitt.

Metod	Förklaring
`add(element)`	Lägger till elementet element sist i listan.
`get(pos)`	Läser värdet hos ett element i position pos i listan.
`add(pos, element)`	Infogar elementet element i den position som anges av pos.
`set(pos, element)`	Infogar elementet element i den position som anges av pos. Tidigare värde ersätts av det nya.
`remove(pos)`	Tar bort och returnerar elementet i positionen pos.
`remove(element)`	Tar bort första förekomsten av element i listan.
`isEmpty()`	Om listan är tom returneras true, annars false.
`clear()`	Tömmer listan på dess innehåll.
`size()`	Ger antalet element i listan.

Låt oss som första exempel se hur metoderna för insättning och hämtning av data kan användas för att fylla en lista med några strängar och därefter skriva ut innehållet till en textruta. För att enkelt få tillgång till listklassen och dess metoder skrivs importsatsen nedan.

```
import java.util.ArrayList;
```

För att skapa en lista med namnet namnLista skriver vi

```
ArrayList<String> namnLista = new ArrayList<>();
```

Lägg märke till att datatypen som ska lagras i listan, i vårt fall String, måste skrivas inom < >-parenteser. En lista kan lagra alla typer av objekt men däremot inte primitiva datatyper som int, double, char och så vidare. Om dessa ska lagras måste de omvandlas till objekt av motsvarande typ. En sådan omvandlling för en int-variabel med namnet heltal kan skrivas enligt satsen nedan.

```
Integer talObjekt = new Integer(heltal);
```

Det får till följd att `talObjekt` blir ett objekt av typen Integer. Detta kan lagras i en lista och vid behov även omvandlas till en variabel av typen int med satsen nedan.

```
int tal = talObjekt.intValue();
```

14.2 Stack

En höstack eller myrstack växer genom att nytt material läggs på toppen. Vill vi minska storleken plockar vi bort material uppifrån om vi inte vill rasera hela stacken. Vid programmering fungerar det på samma sätt. Det är bara den information som senast har placerats i en stack som kan läsas eller plockas bort. Du kan jämföra en stack med en trave tallrikar. Nya tallrikar placeras överst och det är bara den översta tallriken som är åtkomlig. Man säger att en stack fungerar enligt LIFO-principen, *Last In First Out.*

En stack kan betraktas som en lista med begränsningen att data endast kan läggas till på eller tas bort från en enda position, den som kallas stackens topp. Det går alltså inte att infoga eller avlägsna data från någon annan plats i stacken. Trots denna begränsade mängd handlingar är stacken vanlig vid programmering. Orsaken är helt enkelt att det finns så många situationer där lagrad information endast bearbetas med dessa operationer. Ett spelexempel är Black Jack där banken drar kort överst i högen hela tiden.

I tabellen nedan visas några av de metoder som hör till klassen Stack. Flertalet av dessa kommer att förklaras utförligare i detta avsnitt.

Metod	Förklaring
`push(element)`	Lägger till ett element på toppen av stacken.
`peek()`	Läser elementet som ligger på toppen av stacken utan att ta bort det.
`pop()`	Läser och tar bort elementet som ligger på toppen av stacken.
`isEmpty()`	Om stacken är tom returneras `true`, annars `false`.
`clear()`	Tömmer stacken på dess innehåll.
`size()`	Ger antalet element i stacken.

Låt oss som exempel se hur metoderna kan användas för att fylla en stack med städerna Lund, Oslo, Moskva, Madrid och Rom och sedan skriva ut innehållet. För att enkelt få tillgång till stack-klassen och dess metoder skrivs importsatsen nedan.

```
import java.util.Stack;
```

För att skapa en stack med namnet städer skriver vi

```
Stack<String> städer = new Stack<>();
```

14.3 Prioritetskö

Till vardags hamnar vi alltför ofta i en kö. Om vi ska gå på bio står vi i en biljettkö och om vi ringer till banken är risken stor att vi får vänta i deras telefonkö. Köer är även vanliga i datorenas värld. Om du till exempel vill göra en utskrift och din dator är ansluten till en nätverksskrivare placeras ditt dokument i skrivarkön. När ett nytt element ska sättas in i en kö placeras det sist och endast det först insatta elementet kan tas bort. Man säger att en kö fungerar enligt FIFO-principen, *First In First Out*.

I java finns en klass, PriorityQueue, som kan hantera så kallade prioritetsköer. En prioritetskö skiljer sig från en vanlig kö då data ska sättas in i kön. Datapositionen beror nämligen på dess prioritet. Desto högre prioritet data har, desto längre fram i kön infogas det. Strängars prioritet minskar till exempel i alfabetisk ordning. I en prioritetskö som innehåller namn ligger alltså Lisa före Mia eftersom Lisa ligger före Mia i alfabetisk ordning.

I tabellen nedan visas några av de metoder som hör till klassen PriorityQueue. Flertalet av dessa kommer att förklaras utförligare i detta avsnitt.

Metod	Förklaring
`offer(element)`	Lägger till elementet element i slutet av kön.
`peek()`	Läser elementet som ligger först i kön utan att ta bort det.
`poll()`	Läser och tar bort elementet som ligger först i kön.
`isEmpty()`	Om kön är tom returneras `true`, annars `false`.
`clear()`	Tömmer kön på dess innehåll.
`size()`	Ger antalet element i kön.

Låt oss som exempel se hur metoderna kan användas för att fylla en prioritetskö med namnen Lisa, Mia och Jonas och sedan skriva ut dem. För att enkelt få tillgång till klassen och dess metoder skrivs importsatsen nedan.

```
import java.util.PriorityQueue;
```

För att skapa en prioritetskö med namnet personer skrivs

```
PriorityQueue<String> personer = new PriorityQueue<>();
```

Sammanfattning

ArrayList är en klass som bygger ut funktionaliteten för ett fält så att dess längd kan öka och minska dynamiskt, så att storleken inte behöver vara bestämd från början. En kö är en samling data som fungerar enligt principen först in–först ut. En stack är en samling data som fungerar enligt principen först in–sist ut. I Java finns klasserna PriorityQueue och Stack för sådana samlingar.

Metod	Förklaring
`add(element)`	Lägger till elementet `element` sist i listan.
`get(pos)`	Läser värdet hos ett element i position `pos` i listan.
`add(pos, element)`	Infogar elementet `element` i den position som anges av `pos` i listan.
`set(pos, element)`	Infogar elementet `element` i den position som anges av `pos` i listan. Tidigare värde ersätts av det nya.
`remove(pos)`	Tar bort och returnerar elementet i positionen `pos` i listan.
`remove(element)`	Tar bort första förekomsten av `element` i listan.
`isEmpty()`	Om listan är tom returneras `true`, annars `false`.
`clear()`	Tömmer listan på dess innehåll.
`size()`	Ger antalet element i listan.

ArrayList

`push(element)`	Lägger till ett element på toppen av stacken.
`peek()`	Läser elementet som ligger på toppen av stacken utan att ta bort det.
`pop()`	Läser och tar bort elementet som ligger på toppen av stacken.
`isEmpty()`	Om stacken är tom returneras `true`, annars `false`.
`clear()`	Tömmer stacken på dess innehåll.
`size()`	Ger antalet element i stacken.

Stack

`offer(element)`	Lägger till elementet `element` i slutet av prioritetskön.
`peek()`	Läser elementet som ligger först i prioritetskön utan att ta bort det.
`poll()`	Läser och tar bort elementet som ligger först i prioritetskön.
`isEmpty()`	Om prioritetskön är tom returneras `true`, annars `false`.
`clear()`	Tömmer prioritetskön på dess innehåll.
`size()`	Ger antalet element i prioritetskön.

Kö

- ArrayList
- Stack
- PriorityQueue

} klasser