17 Januari 2025

Leon Edlund
19990510-8115
1ME325

Objektorienterad ES5

Introduktion

Denna rapport beskriver utvecklingen av en webbapplikation som simulerar en digital skrivbords miljö i form av ett fönsterbaserat användargränssnitt. Applikationen tillåter användaren att instansiera och använda två olika applikationer, ett tärningsspel samt en klockapplikation som visar den aktuella tiden. Dessa applikationer representeras som separata fönster som kan instansieras flera gånger, stängas ner och flyttas fritt på skärmen med hjälp av musen.

Kodbasen är utformad enligt principer från objektorienterad programmering och följer ECMAScript 5-standarden. Rapporten redogör för hur funktioner används som konstruktörer och hur dessa har strukturerats för att simulera klasser i ett klassbaserat programmeringsspråk. Vidare presenteras implementeringen av arv mellan klasser samt applikationens övergripande struktur. Rapporten diskuterar även vilka åtgärder som har vidtagits för att förbättra minneshanteringen.

Strukturering av klasser

Eftersom kodbasen följer ECMAScript 5-standarden, som saknar stöd för nyckelordet class, har konstruktor-funktioner använts för att simulera klasser. Inom dessa konstruktor-funktioner definieras klassens egenskaper med värden som är unika för det instansierade objektet genom att använda nyckelordet *this*. Nya objekt kan sedan skapas från dessa funktioner med nyckelordet *new* följt av ett anrop till konstruktor-funktionen (Stefanov & Sharman 2013, s. 104).

För att definiera metoder för klasserna har konstruktor-funktionens prototyp använts. Alla objekt i JavaScript har ett tilldelat prototyp-objekt, och eftersom funktioner också är objekt gäller även det för dessa. Eftersom prototypen är ett vanligt objekt går det att lägga till egna metoder och egenskaper som blir kopplade till konstruktor-funktionen och kan nås av alla instanser som skapas av konstruktorn (Stefanov & Sharman 2013, ss. 154-155).

Fördelen med att använda prototyp-objektet är att det endast kräver en minnesreferens. Alla objekt som instansieras från samma konstruktor-funktion delar samma prototyp, vilket innebär att metoder och egenskaper som definieras i prototypen inte behöver dupliceras till varje enskild instans (Stefanov & Sharman 2013, s. 156). Denna teknik har använts i samtliga klasser för att definiera metoder, eftersom dessa är gemensamma och inte behöver vara unika för varje instans.

Arv

Både tärningsspelet och klockapplikationen ärver sin grundläggande funktionalitet för fönsterhantering från en abstrakt klass kallad *CustomWindow* detta görs för att undvika behovet av att upprepa kod för delad funktionalitet mellan olika klasser (Stefanov & Sharman 2013, s. 15). Eftersom ECMAScript-5 saknar inbyggt stöd för arv via ett specifikt nyckelord så används en kombination av två olika tekniker för att uppnå detta, konstruktorstöld och protoypbaserat arv.

Konstruktorstöld används för att få tillgång till basklassens egenskaper. Detta genom att barnklassen kallar på basklassens konstruktor med metoden *call* som binder barnklassens *this* till basklassen och de egenskaper som egentligen skulle initieras i basklassen initieras istället direkt i barnklassen (Stefanov & Sharman 2013, ss. 198-199).

För att få tillgång till basklassens metoder så överskrivs barnklassens prototyp till basklassens prototyp-objekt. För att undvika att basklassen och barnklassen har samma prototypreferens, vilket skulle leda till att ändringar i prototypen skulle påverkar alla klasser som ärver från basklassen, används metoden *Object.create.* Genom att skicka basklassens prototyp som argument till *Object.create* skapas ett nytt tomt objekt vars prototyp pekar på basklassens prototyp (MDN u.å.a). På detta sätt kan barnklassen ärva metoder från basklassens och även definiera egna metoder i prototypen utan att påverka andra klasser som också ärver från basklassen.

Applikations struktur

Applikationen är uppdelade i totalt sju olika klasser: *Main, CustomWindow, DiceApplication, Dice, ScoreCounter, ClockApplication* och *TimeManager.* Nedan följer en beskrivning av varje klass och hur dessa fungerar ihop.

Start av programmet

För att starta programmet har en händelselyssnare kopplats på *window* som initierar *Main* och anropar metoden *start* när hela sidan och alla resurser har laddats in. Detta har gjorts för att säkerställa att allt som krävs för att köra programmet finns tillgängligt innan Javascripten börjar tolkas. För att lämna så få spår som möjligt i det globala omfånget och frigöra minne har egenskapen “once” satts till true vid initiering av händelselyssnaren vilket raderar händelselyssnaren efter att den körts en gång (MDN u.å.b).

Main

Klassen M*ain* är applikationens startpunk och ansvarar för koppling av händelselyssnare för att öppna nya applikationsfönster samt initiering och hantering av dessa fönster.

Klassen *Main* innehåller tre privata egenskaper som refererar till HTML-element i DOM:en. Dessa element inkluderar två knappar som används för att öppna fönster: ett för tärningsspel och ett för klockapplikationen samt en referens till HTML-elementet där fönstren ska läggas till. För att hantera användarinteraktion har händelselyssnare som lyssnar efter klick kopplats till knapparna med metoden *addEventlistners* som introducerades i DOM level 2 event

model, vilket är en del av W3C:s specifikationer för Document Object Model (W3C 2000). Dessa händelselyssnare anropar vardera privat metod som ansvarar för att öppna respektive applikationsfönster genom att skapa en ny instans från rätt klass, som sedan sparas i en statisk array som håller alla öppna fönster.

CustomWindow
CustomWindow är en abstrakt klass vilket innebär att den inte kan instansieras direkt, detta säkerställs genom en kontroll i konstruktorn. Om ett direkt försök görs att skapa en instans av klassen kastas ett felmeddelande. *CustomWindow* är avsedd att fungera som en grund för andra klasser att ärva från, då den tillhandahåller gemensam funktionalitet för fönster som kan återanvändas och utökas för att undvika onödig kod duplicering (Oracle u.å).

Klassens huvudsakliga syfte är att hantera den grundläggande logiken och funktionaliteten för ett fönster i applikationen och används av både *DiceApplication* och *ClockApplication*, men skulle även gå att använda om ytterligare fönsterapplikationer skulle utvecklas.

Klassen hanterar skapandet av den grundläggande HTML-strukturen som omfattar fönstrets ram, menyrad och stängningsknapp där användaren kan skicka med egna CSS-klasser som argument för att erbjuda möjligheten att skapa olika typer av fönster. Metoden *m_CreateWindow* som skapar ett fönster är tänkt att överskridas i barnklasserna för att utöka metoden genom att skapa ytterligare element specifikt för barnklassens struktur.

Klassen innehåller funktionalitet för att stänga ett fönster genom att ta bort fönstret från DOM-strukturen med metoden *remove* på referensen till HTML-elementet (MDN u.å.c). I samband med detta frigörs även resurser genom att ta bort händelselyssnare kopplade till fönstret, anropa *dispose* i underklasserna och sätta värdet på alla egenskaper i klassen till null för att bryta referenser och underlätta för skräphanteraren att rensa minnet.

En annan central funktion är implementeringen av drag-and-drop, som gör det möjligt för användaren att flytta fönster runt på skärmen genom att dra i menyraden med muspekaren. När användaren klickar på menyraden förs fönstret längst fram och lägger sig “ovanpå” resterande fönster. Detta har åstadkommits genom att definiera en statisk egenskap till konstruktorn som initialt får värdet ett, eftersom variabeln är statisk är den kopplad till konstruktorn och är därför inte unik för varje instans. Varje gång användaren klickar på ett fönsters menyrad ökas den statiska egenskapens värde med ett. zIndexet på det klickade fönstret blir sedan uppdaterat till detta värde. Eftersom egenskapen är statisk är det inte kopplat till en specifik instans och det senaste klickade på fönstret får alltid det högsta zIndexet och läggs därför längst fram.

DiceApplication
Klassen *DiceApplication* används för att skapa nya instanser av ett tärningsspel. Klassen ärver sin grund funktionalitet från *CustomWindow.* Konstruktorn tar emot två argument; det maximala antalet tärningar som applikationen har plats att generera, samt en instans av ScoreCounter som representerar och uppdaterar den visuella av poängräknaren.

I klassen överskrids metoden *m_createWindow* från basklassen *CustomWindow* genom att döpa metoden till samma namn. Metoden gör sedan ett superanrop till *m_createWindow* för att köra basklassens metod som skapar grundstrukturen av ett fönster för att sedan skapa de specifika HTML-element för tärningsspelet, såsom knappar och en spelplan. Detta görs för att utöka metodens funktionalitet och anpassa metoden specifikt för tärningsspelet (Stefanov & Sharman 2013, s. 15).

Händelselyssnare kopplas till knapparna som skapats i menyraden av tärnings-applikationen. När användaren klickar på lägga till knappen anropas metoden *m_insertDice* som instansierar ett nytt tärnings-objekt, sparar tärningen i en array och lägger till HTML-elementet från tärnings-objekt på spelplanen för att visuellt visa tärningen för användaren.

För att ta bort en tärning från applikationen så anropas metoden *m_removeLastDice* som tar bort den sista tärningen ur arrayen som håller alla aktiva tärnings-objekt med metoden *pop.* Metoden *pop* returnerar det borttagna objektet som sedan används för att anropa den definierade metoden *delete* i tärnings-objektet (MDN u.å.d).

För att kasta om alla tärningar används metoden *m_reRollAllDice* som går igenom alla instansierade tärningar och anropar metoden *roll* på varje tärning som ger tärningen ett nytt värde och uppdatera CSS-klassen på tränings-elementet. Poängen räknas sedan ihop och poängräknaren uppdateras.

För att kasta om en individuell tärning används en teknik som Stefanov och Sharman (2013, s. 255) kallar event delegation. I stället för att koppla en separat händelselyssnare till varje enskild tärning som anropar *roll*, kopplas istället en enda händelselyssnare till föräldraelementet som innehåller alla tärningar. När användaren klickar på en tärning, används *event.target* för att identifiera vilket element som klickades på (MDN u.å.e). Genom att kontrollera positionen för det klickade elementet i Ul-listan kan tärningens index fastställas och användas för att kasta om motsvarande tärning i arrayen *m_allDice.*

Dice
Klassen *Dice* representerar en enskild tärning och hanterar både skapandet av dess HTML-struktur och logiken för att kasta tärningen genom att genererar ett slumpmässigt värde mellan ett och sex. Efter att tärningen kastats uppdateras tärningens visuella representation genom att dess CSS-klass ändras för att återspegla det nya värdet. Användaren kan kasta om tärningen genom att anropa metoden *roll*, som genererar ett nytt slumpvärde. Den motsvarande CSS-klassen hämtas därefter från en array där det slumpade talet används som index för att korrekt uppdatera det visuella gränssnittet.

ScoreCounter
Poängräknaren som används i tärningsspelet har implementerats som en egen klass för att möjliggöra återanvändning i framtida applikationer. Klassen ansvarar för att skapa och hantera den visuella representationen av en poängräknaren och är designad för att enkelt kunna integreras i olika delar av applikationen.

Räknarens HTML-struktur byggs upp som en ul-lista, där varje siffra representeras av ett eget li-element. Denna struktur sparas i egenskapen *m_element. F*ör att integrera räknaren i en annan del av applikationen kan metoden *getCounter* användas, som returnerar räknaren i form av ett HTML-element och kan där med läggas till på valfri plats i DOM:en.

För att uppdatera räknaren finns den publika metoden *updateCounter*, som tar emot ett nummer (poängen) som kan vara upp till fem siffror långt. Numret konverteras till en sträng, och med hjälp av funktionen *padStart* fylls det på med inledande nollor för att alltid vara fem siffror långt. Därefter itererar en for-loop genom varje siffra i strängen och uppdaterar CSS-klassen för motsvarande li-element i listan. Detta säkerställer att det grafiska gränssnittet återspeglar den korrekta poängen.

ClockApplication
ClockApplication representerar en klocka som visar den aktuella tiden på den enhet som applikationen körs på. Klassen är beroende av klassen *TimeManager* för att fungera då det är *TimeManager* som sköter uppdatering av tid och timingen för uppdatering av klockans siffror. För att undvika att beroendet hårdkodas in i klassen används konstruktionsinjektion. Detta innebär att beroendet injiceras via konstruktorn genom att en instans av *TimeManager* skickas med som argument vid skapandet av en *ClockApplication* instans. Detta tillvägagångssätt möjliggör att *TimeManager* enkelt kan bytas ut vid behov eller testning, vilket resulterar i mer modulär och flexibel kod (Sheldon, 2024).

Även i denna klass överskrids metoden *m_createWindow* från basklassen *CustomWindow* för att utöka metoden och skapa de HTML-element som är specifika för klockan. Varje individuellt-elementen för en siffra i klockan sparas i en array som en egenskap till klassen.

För att uppdatera klockans siffror har en metod skapats som tar emot tiden som en sträng av siffror som argument. Siffrorna itereras sedan igenom där värdet på den aktuella siffran i iterationen används som index för att hämta ut rätt klassnamn ur en array. Denna metod skickas med som argument till *TimeManagers subscribe* metod vilket Stefanov & Sharman (2013, s. 77) kallar en callback funktion.

TimeManager
TimeManager är en singleton-klass, vilket innebär att endast en instans av klassen kan existera. Om flera försök görs att instansiera klassen returneras alltid den först skapade instansen. Detta uppnås genom metoden *getInstance*, som kontrollerar om en instans redan finns sparad statiskt i klassen. Om ingen instans existerar, skapas en ny instans som sedan sparas och returneras. Om en instans redan existerar, returneras den befintliga (Stefanov & Sharman 2013, s. 291). Singleton-mönstret används här för att säkerställa att klockapplikationen som använder klassen får tillgång till samma tidsdata och att alla klockar uppdateras samtidigt.

TimeManager används för att spara och uppdatera den aktuella tiden som en sträng och säkerställa att alla klockapplikationer uppdateras samtidigt och med samma tid.

För att synkronisera alla klockapplikationer används en array som innehåller callback metoder till alla aktiva klockor som uppdaterar klockornas siffror. Dessa callback metoder registreras genom metoden *m_subscribe*, där varje metod sparas i en array. Med hjälp av *setInterval* anropas alla registrerade callback metoder en gång per sekund med den aktuella tiden, vilket säkerställer att alla klockor uppdateras samtidigt.

Minneshantering

För att hantera minnesanvändningen och minska risken för minnesläckor har ett antal åtgärder tagits för att underlätta för skräphanterare att rensa bort minnesreferenser som inte längre behövs. Då Javascript inte har någon funktionalitet för att manuellt rensa minnet som används, finns en automatiskt skräphanterare som automatiskt rensar minnet. Detta görs bland annat genom att kolla om objekt har aktiva referenser till sig, om de inte har det kan minnesreferensen tas bort (MDN, u.å.f).

För att ha möjligheten att ta bort referenser till fönster som stängs så sparas alla referenser till öppna fönster i en statisk array som är kopplad till klassen *Main.* Eftersom arrayen är statisk och inte bunden till en instans går den att komma åt från andra delar av koden. Detta möjliggör att referensen till det aktiva fönstret kan tas bort ur arrayen när fönstret stängs vilket indikerar för skräphanteraren att detta objekt inte längre behövs sparas i minnet.

När fönster stängs tas även alla relevanta händelselyssnare bort, alla egenskaper sätts till null och metoden *dispose* körs i underklasserna som gör en intern rensning i barnklassen. Målet är att ta bort så många referenser som möjligt från objekt för att undvika att onödiga referenser sparas i minnet.

Avslutning

Avslutningsvis har projektet har lett till en bättre förståelse för objektorienterad metodik och hur programkod kan struktureras på ett effektivt sätt för att underlätta arbetet med större kodbaser och minimera kod duplicering, tillexempel genom arv mellan klasser. Det har även varit lärorikt att implementera och praktiskt använda design mönster såsom singelton, abstrakta klasser och överskrivning av metoder i ett verkligt scenario för att se fördelarna med dessa.

Källor

Mozilla Developer Network (MDN). (u.å.a). *Function.prototype.call()*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call [12-01-25]

Mozilla Developer Network (MDN). (u.å.b). *EventTarget: addEventListener() method.* https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener [14-01-25]

Mozilla Developer Network (MDN). (u.å.c). *Element: remove() method.* https://developer.mozilla.org/en-US/docs/Web/API/Element/remove [16-01-25]

Mozilla Developer Network (MDN). (u.å.d). *Array.prototype.pop()*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop [12-01-25]

Mozilla Developer Network (MDN). (u.å.e). *Event: target property.* https://developer.mozilla.org/en-US/docs/Web/API/Event/target [16-01-25]

Mozilla Developer Network (MDN). (u.å.f). *Memory management.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_management [14-01-25]

Oracle. (u.å.). *Abstract Methods and Classes*. https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html [14-01-25]

Sheldon, R. (2024). *What is dependency injection in object-oriented programming (OOP)?* https://www.techtarget.com/searchapparchitecture/definition/dependency-injection [16-01-25]

Stefanov, S., Sharman C. K. (2013). *Object-Oriented javascript,* 2nd Edition. Packt Publishing.

World Wide Web Consortium (W3C). (2000). *Document Object Model Events* https://www.w3.org/TR/DOM-Level-2-Events/events.html [17-01-25]