1. SP13: Aspektno orijentirano programiranje

1.1. Moze li se aspekt atribut definirati i ispred metode, a ne samo ispred klase?

Ne, aspekti su isključivo klase s @*Aspect* atributom.

1.2. Kada je potrebno koristiti cjelokupni AspectJ, odnosno kada nam Spring AOP nije dovoljan?

Ako trebamo savjetovati izvršavanje operacija samo nad “Spring beans”, tada koristimo samo Spring AOP. Spring AOP je jednostavniji od AspectJ. Cjelokupnom AspectJ-u je potreban AspectJ kompajler u build procesu. Ako koristimo objekte koji nisu savjetovani Spring sadržajem, koristit ćemo AspectJ. Source-Qns-8

1.3. Koje su prednosti uporabe AOP naspram obrasca dekoratora?

Primjenom AOP-a je potrebno stvoriti puno manje različitih razreda čime se smanjuje količina pisanja koda i smanjuje se dupliciranje koda. Npr.za logiranje napravimo neki aspekt i za svaku stvar koju želimo logirati stavimo atribut iznad i stvar će se logirati, dok kod dekoratora bi za svaku stvar trebali pisati implementaciju logiranja. Izvor

1.4. Što je aspektno-orijentirano programiranje? Navedi barem tri dostupne implementacije aspektno-orijentiranog programiranja (bilo kao jezik sam po sebi ili kao vanjska biblioteka).

AOP je programska paradigma koja za cilj ima povećanje modularnosti izdvajanjem funkcionalno kohezivnih dijelova koji se odnose na cijelu aplikaciju (*crosscutting concerns*). To postiže tako da se dodaje ponašanje na postojeći kod, bez da se sam kod modificira, tako da se dodaju tzv. “*pointcut*” specifikacije. To omogućava dodavanje stvari koje nisu bitne za logiku, ali se odnose na cijeli program kao npr. logging, bez natrpavanja koda postojeće logike. Najpoznatija je implementacija AspectJ čija je sintaksa *de facto* standard. Od ostalih implementacija možemo još spomenuti Spring, JBoss. Od .NET framwork-a spomenut ćemo PostSharp, Spring.NET...

1.5. Zašto bi trebao koristiti AOP prije dependency injection-a? Korištenje dependency injectiona se ipak drži glavnih OO principa.

Dependency injection i AOP nisu kompetitivne tehnologije, svaka rješava određene probleme i možemo ih koristiti zajedno u projektu. Nikako jedna ne isključuje drugu i zapravo su različite stvari pa ovo pitanje nema smisla.

1.6. Može li se na neki način aspekt definirati unutar tijela neke metode?

Ne, aspekti su isključivo klase s @Aspect atributom. Takva klasa može imati svoje atribute i metode koje nazivamo savjeti (advice).

```
@Aspect
public class TimeLoggingAspect {
}
```

1.7. Povećavamo li korištenjem aspektne paradigme međuovisnost komponenata/klasa? Koje načelo dizaniranja dobrog koda možemo izravno povezati sa aspektnom? (Spearation of Concerns)

Naprotiv, korištenjem AOP smanjujemo međuovisnost komponenata jer smanjujemo količinu koda, a metode koje povezujemo s aspektima nisu “svijesne” tih aspekata pa će se one izvršavati bez obzira postoje li aspekti ili ne - neovisne su. Načelo koje možemo povezati s AOP je Separation of concerns (SoC) koje podrazumijeva razdvajanje programskog koda u distinktne dijelove od kojih svaki obrađuje određeni problem. Još jedno načelo koje možemo spomenuti je izbjegavanje ponavljanja, a to svakako možemo povezati s AOP jer njime smanjujemo količinu koda koji se često ponavlja.

1.8. Objasni kritiku AOP-a da stvara takozvani anti-obrazac akcije na udaljenost?

Anti-obrazac akcije na udaljenost se definira kao obrazac u kojem ponašanje jednog dijela programa bitno varira ovisno o teško ili nemoguće preponatljivim operacijama u drugome dijelu programa. To znači da ako npr. imamo aspekt koji se izvršava nakon npr. svake * set*(*) metode, netko tko čita taj kod neće vidjeti kada pogleda tu set metodu da se nakon nje još *čarobno* izvršava još neki dodatni dio koda pa može pomisliti da je metoda pogrešno implementirana. Zato je ponekad teško pratiti tijek izvršavanja koda.

1.9. Nabroji 3 primjera gdje se često koristi aspektno orijentirano programiranje i razloge zašto.

Uglavnom su to operacije i zadaci koje moramo imati u cijelom programu. AOP se zato često koristi za :

- logging
- caching
- upravljanje transakcijama
- sigurnost
- upravljanje iznimkama i greškama

1.10. Mogu li aspekti savjetovati druge aspekte?

U Spring AOP to nije moguće, dok u AspectJ je.

2. KP4: LINQ

2.1. Npr ako imamo:public static void Disp(this T[] arr){System.out.print(arr.Length*2); } public static void Disp(this String[] arr){System.out.print(arr.Length*3);} I u mainu string[] arr=new string[]{"abc"}; arr.Disp(); Koji Disp se poziva?

Ako pretpostavimo da je kod na koji se mislilo ovaj (jer ovaj gornji nema nikakvog smisla i cijeli je sintaksno pogrešan):

```
public static void Main(string[] args)
        {

            string[] arr=new string[]{"abc"};
            arr.Disp();
        }

public static void Disp<T>(this T[] arr){Console.WriteLine(arr.Length*2); }
public static void Disp(this String[]arr){Console.WriteLine(arr.Length*3);}
```

Poziva se *Disp(this String[] arr)* zato jer se prvo traži metoda s podudarnim tipovima argumenata, a ako takva ne postoji, onda se traži generic.

2.2. Koja je razlika između IQueryable i IEnumerable sučelja?

IQueryable implementira IEnumerable, stoga s IQueryable možemo sve što možemo i s IEnumerable. Inače je IQueryable puno moćniji i koristi se kada npr. pristupamo nekom vanjskom resursu (bazi podataka) zato što se IQueryable upit može prevesti u SQL te zatim provesti nad bazom. IEnumerable će prvo dohvatiti sve redove u memoriju i zatim provesti upit.
Zato je IEnumerable bolji za lokalne kolekcije, a IQueryable za udaljene baze i kolekcije.

2.3. Kad imamo reference na dvije anonimne klase istog prototipa, je li moguće postaviti da jedna referenca pokazuje na drugu, te potom baratati njezinim atributima? Ili je svaka referenca na bilo koju anonimnu klasu sama za sebe specifična?

Sasvim je ispravno napisati sljedeći kod:

```
var a = new {a="1", b="2"};
var b = new {a="3", b="3"};
var c = b;
b = a;
a = c;
Console.WriteLine(a.a);
```

Ali neispravno bi bilo sljedeće:

```
var a = new {a="1", b="2"};
var b = new {x="3", y="4"};
var c = b;

//ovdje bi se dogodila greška o nekompatibilnim anonimnim tipovima
b = a;
```

2.4. Postoji li neka proširena metoda kojom se može zaobići/preskočiti prvih ili zadnjih n rezultata u upitu?

Prvih n elemenata možemo preskočiti pozivom metode **.Skip(n)**, a zadnjih n elemenata možemo zaobići pozivom metode **.Take(lenght - n)**

2.5. Ako postoji kolekcija object-a, mogu li se u nju dodavati anonimni tipovi? Zašto (ne)?

Možemo dodavati anonimne tipove jer se oni prilikom prevođenja tipiziraju u određene tipove koji nasljeđuju klasu *object.*

```
List<object> listOfObjects = new List<object>();
var item1 = "Ja sam string";
var item2 = 3;
var item3 = new {atribut="1"};
listOfObjects.Add(item1);
listOfObjects.Add(item2);
listOfObjects.Add(item3);
foreach (var obj in listOfObjects){Console.WriteLine(obj);};
```

Ispis: Ja sam string
3
{ atribut = 1 }

2.6. Koji je razlog korištenja LINQ upita naredbama (in, from, select...) naspram metoda proširenja (.Select(), .Where() ...) ?

Ne postoji velika razlika u korištenju jer se prevode(kompajliraju) jednako. Za kraće upite, metode proširenja mogu biti kompaktnije, a također mogu biti i fleksibilnije. Neke metode poput Skip, Take, ToList, itd. su dostupne samo u metodama proširenja pa ako trebamo koristiti neku od tih metoda, držat ćemo se metoda proširenja. No, ako upit postane veći i kompleksniji, onda su nam upiti naredbama jasniji i čitljiviji. Upit naredbama je zapravo "*sintaksni šećer*" za programere koji su naučeni na SQL.

2.7. Koja je prednost koristenja LINQ nad pohranjenim procedurama?

U LINQ možemo koristiti lambda izraze, možemo iskoristiti iste upite za razne izvore podataka (baze, kolekcije, xml, itd.), imamo korisne metode kao .Skip() i .Take(). LINQ upiti su puno čitljiviji od upita pohranjenih procedura, a time je kod i lakši za održavanje.

2.8. Dohvaća li query varijabla podatke odmah pri stvaranju varijable ili u nekom drugom trenutku?

Stvaranjem query objekta ne dohvaćaju se podatci. Query objekt zna kako dohvatiti podatke, ali ne zna same podatke. Query dohvaća podatke kada je enumeriran (npr. kada se koristi u foreach petlji).

2.9. Je li moguće izvršiti sljedeći dio koda te ako nije, što treba promijeniti? var customers = new[]{ new { Name = "Marco", Discount = 2.5 }, new { Name = "Paolo", Discount = 3.0 } }; IEnumerable(string) customerNames = from customers as c where c.Discount > 3 select new { c.Name };

Ispravan upit bi bio :

```
IEnumerable<string> customerNames =
      from c in customers
      where c.Discount > 3
      select c.Name;
```

Zato što C# ne može implicitno prebaciti anonimni tip u string, pa smo odmah u select tijelu uzeli string.
Također smo mogli iskoristiti ***var*** tip, pa kasnije koristiti string varijablu

```
var customerNames =
      from c in customers
      where c.Discount > 3
      select new {c.Name};
foreach (var c in customerNames)
{
Console.WriteLine(c.Name);
```

```
};
```

2.10. Da li se može, i ako da , kako proslijediti anonimne tipove drugim funkcijama?

Odgovor je, naravno, pozitivan. Ilustrirat ćemo tu mogućnost sljedećim isječkom koda. Koristimo ključnu riječ ***dynamic*** pri definiciji tipa argumenta. Tada se tip određuje tijekom izvođenja, a ne tijekom prevođenja. Ako predani objekt nema atribut "name" doći će do iznimke.

```
public static void Main(string[] args) {
	var item = new {name="Marko", surname="Polo"};
	testPrint(item);
}

public static void testPrint(dynamic arg) {
	Console.WriteLine(arg.name);
}
```

Ispis će biti: Marko

2.11. Na koji način metoda proširenja, koja kao argument prima polje podataka, može postati univerzalna za sve tipove polja?

Tako da se proširi polje elemenata generičkog tipa. Primjerice ako koristimo generčku varijablu T za tip polja:

```
public static T[] Extend<T>(this T[] array, T item)
{
	// kod
}
```

Restrikcije na tip se može uvesti pomoću "where T : [neki tip]".

Slično ponašanje se može postići tako da se uvede proširenje za polje objekata (budući da svi tipovi u C# nasljeđuju System.Object), no tada bi se trebalo prije korištenja metode castat polje u polje objekata te castat nazad nakon izvođenja operacije, dok se generički tip pri kompilaciji dodjeljuje tip metodi koji je potreban i ostvaruje polimorfizam

2.12. Kako se u LINQ-u postize left outer join, a kako inner join?

Left outer join (uzme sve atribute s lijeve strane, a ako ne postoje s desne postavlja na null):

```
var query = from item1 in collection1
            join item2 in collection2
            on collection1.id equals collection2.id
            into jointCollection
            from item in jointcollection.DefaultIfEmpty()
          select new {id=collection1.id, name=collection2.name,...}
```

Inner join (uzima samo elemente koji imaju vrijednosti s obje strane):

```
var query = from item1 in collection1
            join item2 in collection2
            on collection1.id equals collection2.id
            into jointCollection
            where jointCollection.Any()
          select new {id=collection1.id, name=collection2.name,...}
```

2.13. U C# projektu postoji klasa s konstruktorom "public AB(int a, float b)". U metodi tog projekta postoji polje "var polje = new[] {new { A = 1, B = 1.0f, C = "1" }, ...};". Nadopuni kod "IEnumerable rezultat = ___;" LINQ upitom tako da u variabli rezultat budu samo elementi kojima je C jednak "1".

```
IEnumerable<object> rezultat =
      from el in polje
      where el.C.Equals("1")
      select el;
```

3. SP15: Funkcijsko programiranje u F#

3.1. Koje su prednosti funkcijskog programiranja u odnosu na objektno orijetirano programiranje?

Funkcijsko programiranje je bolje kada imamo stalni set podataka i trebamo nad njime provoditi puno različitih operacija, dok je OOP bolje za različite tipove podataka koji se različito ponašaju. Zbog svojstva nepromijenjivosti, funkcijsko programiranje je bolje za paralelizme, a također je zbog nepromjenjivosti olakšan posao “garbage collectoru” pa može značiti bolje upravljanje memorijom. FP je modularno s obzirom na funkcionalnost, a OOP je modularno s obzirom na svojstva. Zbog načela “čistog (pure) funkcijskog programiranja” možemo sigurno i bez straha zvati funkcije nekoliko

puta za redom, čak i iz drugih dretvi. FP je zbog toga i bolje za testiranje - funkcije se ponašaju kao matematičke funkcije: “Ako daje ispravan rezultat danas, davat će ispravan rezultat i sutra!”.

3.2. Budući da je F# primarno funkcijski jezik, postoje li u njemu uopće varijable ili su one zapravo funkcije koje ne primaju nijedan argument i kao povratnu vrijednost vraćaju onaj iznos koji je konceptualno dodijeljen toj "varijabli"?

U F# zapravo ne postoje varijable, nego ih nazivamo vrijednostima (values) zato što se one kompajliraju kao statičke i read-only vrijednosti i kada jednom pridodamo "varijablu" nekoj vrijednosti, ona se ne mijenja nikada.
Primjer za to je :
let x = 10
Dalje više nećemo moći mijenjati vrijednost x.

Ipak, možemo kombinirati kada ćemo tu vrijednost spremiti i "izračunati":
let x //evaluira se na mjestu definicije
let x() //evaluira se prilikom svake upotrebe
let x = lazy(f(y)) //evaluira se samo jednom, prilikom prvog korištenja

3.3. Što je to "lazy computation" u F#?

"Lazy computation" je svojstvo jezika da izračunava vrijednosti samo onda kada je potrebno, a "preskače" izračunavanje kada je ono nepotrebno.
Primjer (pseudo-kod):
funkcija():print("Pozvana je funkcija"); return 1;

print(funkcija());
print(funkcija());

Ispisat će se:

> ***Pozvana je funkcija***
> ***1***
> ***1*** //Drugi put se ne poziva funkcija jer je već poznata njezina povratna vrijednost

3.4. Napišite kod u f# koji stvara listu brojeva, funkciju koja provjerava proste brojeve u listi, i liniju koda koja filtrira listu po toj funkciji i zatim iterira ispis članova

```
let genRandomNumbers count =
      let rnd = System.Random()
      List.init count (fun _ -> rnd.Next ())
let isPrime n =
      let rec check i = //rec je keyword za rekurzivnu fju
           i > n/2 || (n % i <> 0 && check (i + 1))
      check 2
let brojevi = genRandomNumbers 100 //generiramo 10 random
brojeva let listPrime lst =
      lst |> List.filter isPrime //filtriramo proste
brojevi |> listPrime |> printfn "%A" //ispis
```

3.5. Što je 'self' u F#-u?

Self identifikator je referenca na trenutni objekt. Ekvivalentan je ključnoj riječi "this" u Javi i C#. Možemo ga nazvati kako god želimo u F#.

3.6. Što je to čista funkcija (pure function)?

"Pure" funkcija je ona funkcija na čiji rezultat utječu samo ulazne vrijednost, a njezin utjecaj na "okolinu" je samo povratna vrijednost bez ikakvih dodatnih efekata.
Primjer (**pseudo-kod**):
//pure
fukcija(a, b): return a + b;

//nije pure jer mijenja dio okoline (varijablu x)
funkcija(x) : return x+=1;

3.7. Stvorimo varijablu "let myInt = 5". I F# prepozna da je to integer. I onda od te varijable oduzmemo 0.01. "myInt = myInt - 0.01". Hoće li tada F# promijeniti svoje mišljenje iz integera u real i izvesti operaciju ili će izbaciti error?

Dobili bismo pogrešku o različitim tipovima:
"The type 'float' does not match the type 'int'"
Ali zato što su varijable u F# "immutable", da smo pokušali i "myInt = myInt + 10" dobili bismo sljedeću pogrešku:
"Duplicate definition of value 'myInt'"

3.8. Pošto funkciju s parametrima u F# zovemo nazivom funkcije te zatim parametre navodimo razmacima, kako bi predali kao parametar funkciju s parametrima?

Normalnim pozivom tako da nakon parametra koji je ime funkcije navedemo njene argumente:

```
let add a b = printf "%d" (a+b)
let ispis f = f
ispis add 1 2
```

Ako želimo predati i neke dodatne argumente, stavimo poziv funkcije s argumentima u zagradu:

```
let add a b = printf "%d\n" (a+b)
let ispis f x y = f; printf "%s\n" x; printf "%s\n" y;
ispis (add 1 2) "argumenti" "funkcije"
```

Ispisat će se:
3
argumenti
funkcije

3.9. Koja je razlika između tuplea, listi i polja u F#?

Tuple je poredana n-torka elemenata koji mogu biti različitog tipa.

```
let mojTuple = (“string”, 4, 4.3)
```

Lista je skup elemenata istog tipa i u nju uvijek možemo dodavati nove elemente.
Polje (eng. array) je slično listi, ali je unaprijed određeno brojem elemenata i tipom (odnosno određeno je veličinom koju zauzima u memoriji). Pogodno je za spremanje velikih količina elemenata primitivnih tipova.

3.10. Zašto je nepromjenjivost jedno od važnijih svojstava u funkcijskom programiranju?

Nepromjenjivost ili immutability je važno svojstvo koje znači da su objekti nepromjenjivi nakon što ih kreiramo. Ono nam omogućava sigurnost podataka u višedretvenim aplikacijama, a funkcijsko programiranje nas zbog toga “tjera” da pišemo višedretveno sigurne programe. Zbog toga ne moramo brinuti o side-efektima funkcija, jer su oni nemogući.

3.11. Koja je razlika između List i Sequence? Koje je bolje koristiti i kada?

Sequence je pandan IEnumerable<T> u C#, znači da se evaluira tek kada počnemo iterirati kroz nj (“lazy evaluation”). Efikasnije koristi memoriju od liste jer se u jednom trenutku samo jedan element učitan. Mana je što možemo iterirati samo “prema naprijed”, a traženje po indeksu je sporo. ***Liste*** su najbolje za manje količine podataka. Nad njima možemo provoditi kompleksne iteracije pomoću rekurzija.

3.12. Čemu služi "unit" tip u programskom jeziku F#?

“unit” tip je tip u F# koji je ekvivalentan tipu “**void**” npr. u C# ili C++. Često ga koristimo kada je neka vrijednost potrebna zbog sintakse jezika, ali nama nije potrebna ili poželjna. **unit** nema vrijednost, nego je označen kao prazni tuple “**( )**”.

3.13. Za naredbe u F#-u: let neg x = x *(-1) let square x = x*x, Što se dobije ispisivanjem neg >> square ?

“**>>**” je operator za kompoziciju funkcije. Znaci da će se prvo izvesti funkcija ***neg*** pa će se njen rezultat predati funkciji ***square***.
Primjer:

```
let neg x = x *( -1)
let square x = x*x
let comp = neg  >> square
printf "%d" (comp 3)
//prvo mnozimo 3 s (-1) pa kvadriramo, ispisat će se 9
```

4. SP16: Stat i din upravljanje memorijom 1

4.1. Što je krivo u navedenom isječku koda? int main() { int *pointer = (int *)malloc(sizeof(int)); pointer = NULL; free(pointer); }

Nećemo osloboditi memoriju koju smo alocirali u prvoj liniji main fukncije, jer smo linijom ***“pointer=NULL;***” izgubili referencu na alociranu memoriju. Moramo prvo izvršiti free naredbu, a zatim je postavljanje pointera na NULL samo dobra, ali neobavezna praksa.

4.2. Na koji način operacijski sustav upravlja sa "rupama" na heap-u? Na primjer, ako nema dovoljno velike rupe u memoriji, koju je korisnik zatražio, hoće li ono realokacijom ostalih objekata napraviti dovoljno veliku rupu ili će odbiti alokaciju?

Malloc će u tom slučaju vratiti NULL pointer, što znači da alokacija nije uspjela. Programer se mora sam brinuti da ne dođe do fragmentacije i ovakvih problema povezanih s fragmentacijom. Iznimka su programski jezici s implementiranim garbage collectorom, kod kojih ne trebamo brinuti o tome.

4.3. Postoje li mehanizmi za ručno oslobadjanje memorije u jezicima koji imaju ugrađen Garbage Collector? Ako da, navedi primjer kada je dobro taj pristup koristiti.

Postoje. U mnogim jezicima možemo ručno pozvati garbage collector kada nam zatreba.
Primjerice u Pythonu:

```
import gc
gc.collect()
```

Slično u C#:

```
GC.Collect()
```

Primjer u Javi:

`System.gc()` //ne poziva se GC, već se samo daje prijedlog virtualnoj mašini, koja to može i ignorirati.

Primjer korištenja bi bio prilikom debuggiranja, kada forsiranje garbage collectora može otkriti curenje memorije.
Inače ručno pozivanje GC-a nije dobra praksa jer može bitno utjecati na performanse programa.

4.4. Može li se garbage collector ručno pozivati u jezicima koji ga koriste?

U nekima se može pozvati ručno(Python, C#), u nekima se može dati naputak da se pokrene, ali ga compiler/virtualna mašina može ignorirati(Java). Inače ručno pozivanje GC-a nije dobra praksa jer može bitno utjecati na performanse programa.

4.5. Navedi primjer u kojem je bolje statičko upravljanje memorijom od dinamičkog i primjer u kojem je dinamičko upravljanje memorijom bolje nego statičko.

Ako pišemo velik i kompleksni program za koji je bitan svaki segment i ponašanje u memoriji, odabrat ćemo statičko upravljanje (C, C++) zato što imamo veću slobodu i moć prilikom optimizacije programa. No programer mora biti vješt jer iako je ovo moćan način upravljanja memorijom, on sa sobom nosi i mnoštvo neželjenih problema poput fragmentacije i korupcije memorije itd.
Ako pak pišemo program za koji nije toliko da bude najoptimiziraniji jer nije toliko kompleksan, koristit ćemo dinamičko upravljanje memorijom i sve prednosti jezika koji imaju ugrađen garbage collector. Često su i jezici s dinamičkim upravljanjem memorijom apstraktniji i lakši za shvatiti ljudskome umu.

4.6. Što se dešava kada oslobodimo blokove memorije različite veličine razlčitim redosljedom?

Dolazi do fragmentacije, odnosno do “rupa” u memoriji do kojih ne možemo doći (alocirati). Tu memoriju ne trošimo, ali ona je nedostupna pa naš program zauzima više memorije nego što bi trebao.

4.7. Na koji način je moguće u C++ rješiti problem curenja memorije? Na koji način je potrebno modificirati normalne pokazivace kako bi to ostvarili? (Smart pointers)

Problem curenja memorije možemo riješiti na nekoliko načina. Jedan od njih je svakako korištenje “smart pointera”. Radi se o omotačima (wrapper) oko *sirovih* pokazivača koji se brinu o oslobađanju iz memorije umjesto nas. Primjer za to je tzv. “*shared pointer*” koji možemo primjeniti u nekom objektu koji ima pointer na neki drugi objekt. Kada uništimo prvi objekt, *shared pointer* će se pobrinuti da se i drugi objekt oslobodi iz memorije. Uz to se preporuča korištenje *RAII* principa - *“Resource acquisition is initialization”*.

4.8. Postoji li alternativni nacin oslobadanja memorije osim operatorom delete?

Postoje dva načina za oslobađanje memorije. Operator delete koristimo kada je memorija alocirana operatorom new. Drugi način je operatorom free(), a njega koristimo kada smo alocirali memoriju naredbom malloc. Delete operator pozvat će destruktor objekta koji pokušavamo osloboditi, dok free neće. Alternativni načini oslobađanja memorije su npr. razne implementacije pametnih pointera od kojih će se neki čak znat sami osloboditi ovisno o samoj implementaciji(slično garbage collectoru).

4.9. double* p = new double{11.}; p = new double{12.}; delete p; - Kako se naziva problem koji nastaje u ovom odsječku? Koji je ispravan način za brisanje pokazivača?

Dolazi do “memory leak-a”, odnosno curenja memorije jer smo alocirali memoriju za “new double {11.}” i nakon toga izgubili referencu na taj dio memorije. Ispravno rješenje:

```
double* p = new double {11.};
delete p;
p = NULL;
p = new double {12.};
delete p;
p = NULL;
```

Ovdje također možemo spomenuti korištenje klasa pametnih pointera kao rješenje, koji bi se sami pobrinuli za dealociranje starog objekta.

4.10. Koja je razlika između spremanja objekta na stog i na gomilu? Jesu li veličine spremljenog objekta jednake? Zašto?

Objekt na stogu stoji sve dok ne izade iz djelokruga (scope), npr. nakon izvršenja funkcije u kojoj je stvoren, objekt se briše. Na gomili objekt će ostati sve dok ga programer ručno ne oslobodi. Stog ima svoju veličinu i ako dodamo dovoljno elemenata na nj, može doći do stack overflowa. Na heapu do toga ne može doći jer OS može povećati heap ako i kada je to potrebno. Stog je puno brži od heapa.

5. SP17: Stat i din upravljanje memorijom 2

5.1. Na koji način substring metoda u Javi može uzrokovati curenje memorije?

Ovaj problem je bio moguć u Javi sve do verzije 7u6 kada je popravljen.
Problem je bio u tome što je Java ostavljala referencu na originalni ***string*** kako bi imala pristup traženom ***substringu***, stoga je bilo moguće da substring od jednog znaka drži referencu na jako dugački znakovni niz, time uzrokujući memory leak.
Problem je bilo moguće riješiti koristeći konstruktor za novi string:

```
String s2 = new String(s1.substring(0,1));
```

Od Jave 7u6 ovo je riješeno, pa substring vraća sasvim novi string, izbjegavajući time mogući memory leak.

5.2. Kada objekt postaje podložan garbage collectionu? Tj, kako garbage collector zna koje objekte smije "odbaciti" ?

Kada više ne postoji niti jedna referenca (pointer) na taj objekt, on postaje nedohvatljiv i tada ga garbage collector može skupiti**.** To se može dogoditi kada objekt izađe iz djelokruga (scope), kada eksplicitno postavimo pointer na null vrijednost ili kad reinicijaliziramo objekt.

5.3. Koje dvije vrste fragmentacije postoje i koje su njihove razlike?

Unutarnja i vanjska fragmentacija.
Unutarnja fragmentacija - za proces se alocira više memorije nego je potrebno pa se ona ne može koristiti za druge
Vanjska fragmentacija - između dva bloka alocirane memorije ostaje prazne memorije koja je premala za alokaciju novog objekta pa propada; u memoriji postoji dovoljno prazne memorije za alokaciju ali ona nije iskoristiva jer je rascjepkana

5.4. Što je to fragmentacija i kako dolazi do nje?

Fragmentacija je problem koji se očituje u tome da imamo slobodne memorije, ali ona nije objedinjena pa stvara problem pri alociranju nove memorije. To je vanjska fragmentacija, postoji i unutarnja koja se javlja kada alociramo previše memorije pa se ona ne može iskoristiti za daljnju alokaciju. Do fragmentacije dolazi nepažljivim oslobađanjem i realociranjem te neracionalnim alociranjem memorije.

5.5. int i; int main() { int j; int *k = (int *) malloc (sizeof(int)); } Gdje su u memoriji pohranjene varijable i, j, k i zašto?

-i je globalna varijabla i ona je zato spremljena u data segmentu aplikacije, a životni vijek joj je isti kao i aplikaciji

-j je varijabla pohranjena na stogu (stack), njena velićina je poznata prilikom kompajliranja, ali se memorija alocira tek prilikom pokretanja programa

-k je dinamički alocirani pointer (pokazivač), i on pokazuje na memoriju na heapu (gomili)

5.6. Napiši C++ odsječak koda koji će osloboditi memoriju zauzetu na sljedećem odsječku koda (uz predpostavku da imate pristup variabli "A* a"): "a = new A(); a->b = new B(); a->b->c = new C();"

```
delete a;
```

Operator ***delete*** će osloboditi **a** iz memorije, ali ne znamo što se događa s **b** i **c** jer nemamo dostupne njihove destruktore. Rješenje za to bi bilo da iskoristimo *smart pointer*:

```
class A { std::shared_ptr<B> b; }
```

Nakon toga defaultni destruktori će se pobrinuti da se oslobode b i c iz memorije kada pozovemo delete.

5.7. Kada C# pokreće garbage collection?

U nekoliko slučajeva se pokreće:

- Kada nije više moguće alocirati/realocirati memoriju, GC se pokreće kako bi se oslobodilo što više memorije
- Kada se alocira veća količina memorije (npr. 1MB) pokreće se GC
- Ako se ne uspije alocirati neki nativni resurs
- Kada ručno pozovemo GC
- Kada prebacimo aplikaciju u pozadinu

5.8. U javi naredbom System.gc() mozemo dati hint JVM da pokrene garbage collector koji se ne mora odmah pokrenuti. Zasto nemamo punu kontrolu nad garbage collectorom?

Smisao GC-a je da se programer ne mora brinuti o upraljavnju memorijom, zato i postoji GC koji u zasebnoj dretvi odraduje kompleksne zadatke i sam određuje koje objekte može osloboditi iz memorije. Često je ta operacija resursno skupa, pa može utjecati na performanse. Zato nije dana kompletna kontrola programeru, jer sama dretva zna najbolje odrediti kada će, što i kako osloboditi iz memorije, a da to bude korisno za program.

5.9. Ako klasa napisana u c++ sadrži varijable int a i float b, gdje će biti alocirana memorija i koliko B će iznositi ako smo objekt stvorili sa ključnom riječi new ?

Memorija će biti alocirana dinamički, odnosno sprema se na ***heapu.*** Ovisi o arhitekturi operacijskog sustava, na 32-bitnom je veličina pokazivača 4B pa će ukupno zauzeće biti 8B, na 64-bitnom veličina pokazivača je 8B pa će biti 16B.