OSNOVNI GRADNIKI PROGRAMSKIH JEZIKOV 2. DEL

Programiranje 1 - RIN, BF

Računalništvo 1 - MA, MF

Predavatelj: Niki Hrovatin

Vsebina

Objektno usmerjeno programiranje (OOP)

- Abstrakcija za obvladovanje kompleksnosti
- Trije načela OOP

Razredi

- Deklaracija – struktura
- Spremenljivke – lastnosti
- Funkcije – metode
- Konstruktorji
  - Inicializacija

Predmeti

- Dostopnost
- Življenjski cikel predmeta:
  - Kako predmet ustvarimo, uporabljamo, uničimo?
  - Garbage collector (JAVA)

Podajanje argumentov

Podpisi funkcij in pogodbe

Objektno usmerjeno Programiranje - OOP

Vsi računalniški programi so sestavljeni iz dveh elementov: **koda** in **podatki**.

- Programe lahko organiziramo okoli kode ali okoli podatkov.

Procesno usmerjen model prvi programski jeziki, kot je C

- Program je bil serija **linearnih korakov**, ki delujejo na podatkih.
- Velik program postane zelo kompleksen
  - težko ga je spreminjati, vzdrževati in odpravljati napake.

Objektno usmerjeno programiranje (OOP)

- Organizacija programa okoli **podatkov** z uporabo **objektov** / **predmetov**.
- Ponuja način obvladovanja kompleksnosti programov z **abstrakcijo**.

Abstrakcija za obvladovanje kompleksosti

Definicija: *Konceptualna abstrakcija se nanaša na filtriranje informacij o konceptu, pri čemer izberemo le tiste vidike, ki so pomembni za določen namen.*

Naraven način, kako ljudje obvladujejo kompleksnost.

Primer avtomobil:

- Ljudje ne razmišljajo o avtomobilu kot o velikem številu delov, ki skupno delujejo.
- Vidijo ga kot enoten objekt s specifičnim vedenjem.
- Ni potrebno razumeti, kako deluje motor ali zavore — samo pospešujemo, zaviramo in krmilimo, pri čemer ignoriramo ostalo.

Abstrakcija za obvladovanje kompleksosti

Hierarhična abstrakcija

- Razslojevanje kompleksnih sistemov na bolj obvladljive dele.

Primer avtomobil:

- Od zunaj je avtomobil enoten objekt.
- Ko pa pogledamo v notranjost, vidimo, da je sestavljen iz podsistemov:
  - volan, zavore, zvočni sistem, varnostni pasovi itd.
- Vsak podsistem je sestavljen iz bolj specializiranih enot — kot je zvočni sistem, ki lahko vključuje radio, CD-predvajalnik ali MP3-predvajalnik.

Trije načela OOP - Enkapsulacija

Enkapsulacija

- Je načelo združevanja **podatkov** in **kode**, ki z njimi upravlja, v eno celoto, pri čemer nadzorujemo, kako lahko do podatkov **dostopamo** in jih **spreminjamo** od zunaj.
- Organizacija in zaščita podatkov.

Eden od načinov, kako razmišljati o **enkapsulaciji, je kot o zaščitni ovojnici**

- Preprečuje, da bi bila koda in podatki dostopni z drugo kodo, definirano zunaj ovojnice.
- Dostop do kode in podatkov znotraj ovojnice je nadzorovan preko dobro definiranega **vmesnika**.

Trije načela OOP - Dedovanje

Dedovanje

- Je mehanizem, ki razredu omogoča, da **podeduje lastnosti in vedenje drugega razreda**, kar omogoča ponovno uporabo kode in ustvarjanje bolj specializiranih razredov.
- Ponovna uporaba in specializacija razredov.

Omogoča hierarhično abstrakcijo.

- Razrede lahko organiziramo v **hierarhije**, od splošnejših do bolj specifičnih tipov.

Trije načela OOP - Polimorfizem

Polimorfizem

- Polimorfizem (iz grščine, pomeni »mnogo oblik«).
- Je lastnost, ki omogoča uporabo **istega vmesnika** za različne vrste dejanj.
- **Katero dejanje se dejansko izvede**, je odvisno od konteksta.

npr:

- Tako `Car` kot `Boat` imata lahko metodo `move()`, vendar se avtomobil premika po cesti, čoln pa po vodi. Vsak objekt izvaja isto metodo na svoj način.

Predmeti oziroma stvari

Predmeti se razlikujejo po svojih lastnosti.

- Lastnosti tudi določajo, kaj lahko s predmeti počnemo — imamo metode za **spreminjanje** lastnosti in **poizvedovanje** o njih.

Predmeti:

- Hranijo **podatke**: vrednosti lastnosti; npr. lastnost **barva** lahko ima vrednosti, kot so zelena, modra, rdeča itd.
- Nudijo **metode** za delo s temi podatki; npr. `pobarvaj`.

Razredi

Razredi se uporabljajo za definiranje novega **podatkovnega tipa**.

- Novi tip se uporablja za ustvarjanje **predmetov** tega tipa.

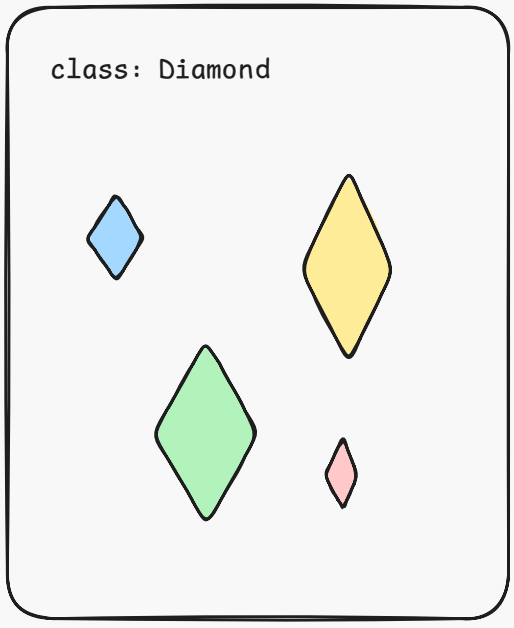
Tako je razred *predloga za predmet*, in **predmet je instanca razreda**.

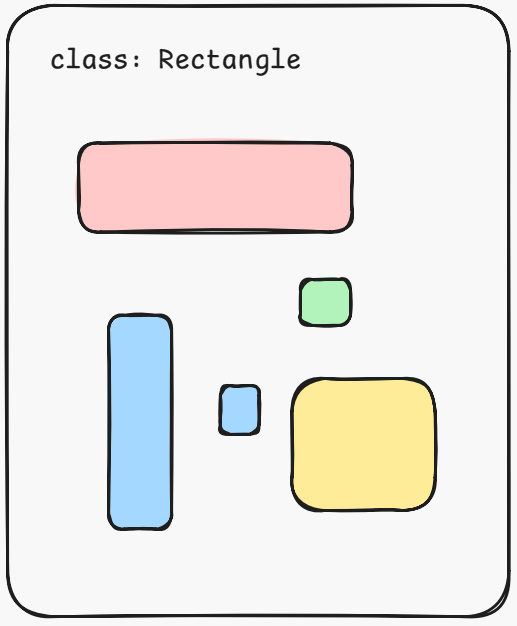
- Ker je predmet instanca razreda, pogosto vidimo, da se izraza **predmet** in instanca uporabljata **zamenljivo**.

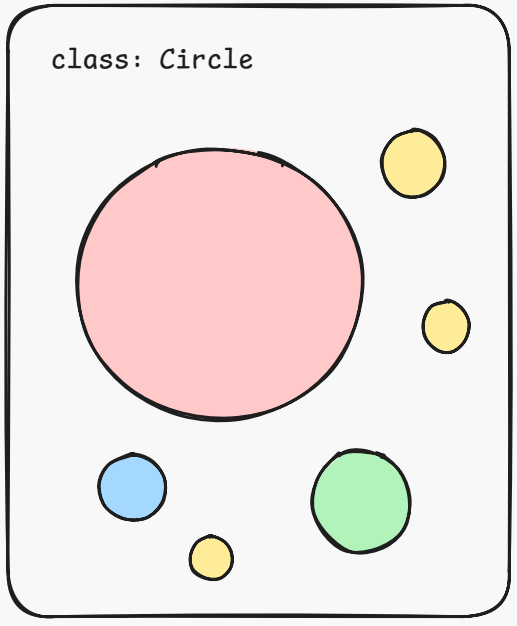
Razred kot predloga za predmete

- Vsi predmeti v razredu si delijo iste lastnosti; na primer, vsi lahko imajo lastnost barve.
- Vendar pa nimajo nujno vsi predmeti v razredu enake vrednosti za to lastnost; na primer, ni nujno, da so vsi predmeti zeleni.

Razredi



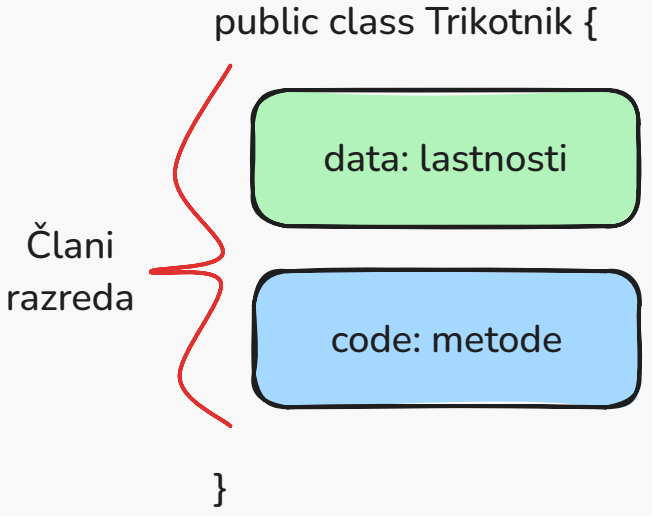




Razredi

Razred v Javi je način predstavljanja **koncepta**.

- To storite tako, da določite značilnosti koncepta:
  - **Podatke**, ki jih vsebuje
  - **Kodo** / **funkcionalnost**, ki deluje na teh podatkih.



Značilnosti koncepta == **člani razreda**

- podatki == **lastnosti**
- koda / funkcionalnosti == **metode**

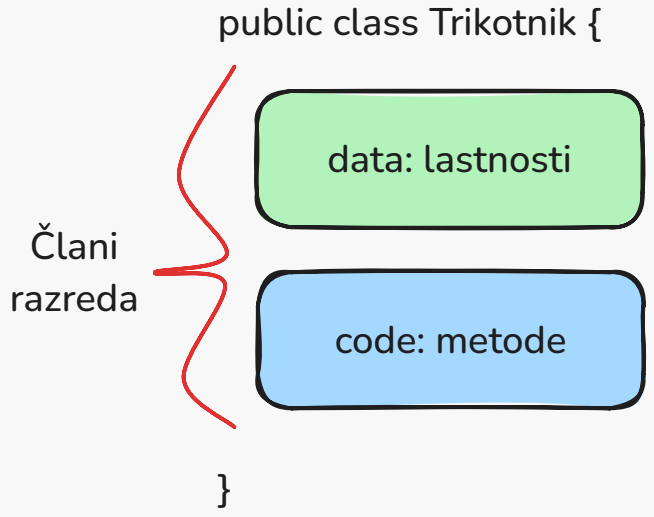
Razredi

Podatki

- Trikotnik: podatki so velikost stranic, barva itd.

Koda

- Koda predstavlja vedenje trikotnika
- Trikotnik lahko izračuna površino, obseg, se lahko premakne itd.



Razredi

Poimenovalna konvencija

- `CamelCase` - vsaka beseda se začne z veliko začetnico
- Opisna in na samostalnikih osnovana imena
- Izogibajte se kraticam, uporabite `CustomerAccount` namesto `CustAcc`

Kategorizacija metod razreda

- Ustvarjanje / Uničevanje
- Poizvedovanje
- Spreminjanje
- Zasebne lastnosti in metode (notranje)

Razredi

V programiranju najprej definiramo (opišemo) značilnosti razreda predmetov — definiramo razred.

Nato ustvarimo eno instanco razreda — ustvarimo predmet.

Primer: Ustvarimo razred za upravljanje trikotnikov

- Trikotniki bodo imeli stranice in barvo
- Trikotniki bodo omogočali izračun obsega in površine
- drugo ...

Razredi - declaracija

Deklaracija lastnosti:

```
public class Trikotnik {

    int a,b,c;
    String barva;

}
```

Ustvarjanje predmetov -- new operator

Operator new dinamično dodeli (med **izvajanjem**) pomnilniški prostor za predmet in vrne referenco nanj.

- Referenca je naslov v pomnilniku kjer se nahaja predmet, ki ga dodeli new.
- Dodelitev med izvajanjem zagotavlja, da program lahko ustvari toliko predmetov, kolikor jih potrebuje.

Ustvarjanje predmetov:
Dvofazni postopek.

1. Deklarirajte spremenljivko tipa razreda.
2. Ustvarite predmet in njegovo referenco dodelite spremenljivki.

```
Trikotnik tri;          // deklaracija
tri = new Trikotnik(); // instanciranje
```

Dostop do lastnosti predmeta

Za dostop do lastnosti predmeta uporabite operator (.)

```
public static void main(String[] args) {
    Trikotnik tri = new Trikotnik();
    tri.barva = "rdeča";
    tri.a = 5;
    tri.b = 6;
    tri.c = 7;
    System.out.println(tri.barva);  // rdeča
    System.out.println(tri.a);      // 5

    Trikotnik tri2 = new Trikotnik();
    tri2.barva = "zelena";
    tri2.a = 10;
    System.out.println(tri2.barva);  // zelena
    System.out.println(tri2.a);      // 10
}
```

Spremembe enega predmeta nimajo vpliva na druge predmete.

Konstruktor

Konstruktor inicializira predmet takoj ob ustvarjanju.

- Ima enako ime kot razred
- Navadno je public
- Kliče se pred dokončanjem operatorja new
- Nima povratnega tipa (implicitno konstruktor vrne tip razreda)
- Inicializira notranje stanje predmeta

Klic privzetega konstruktorja:

- Vsak razred vedno ima privzeti konstruktor

```
Trikotnik tri = new Trikotnik();
```

Konstruktor

```
Trikotnik(){
    System.out.println("Ustvarjam trikotnik");
}
Trikotnik(){
    System.out.println("Ustvarjam trikotnik");
    a = 5;
    b = 6;
    c = 7;
}
```

Parametriziran konstruktor:

```
Trikotnik(int stranA, int stranB, int stranC, String barvaTrikotnika){
    System.out.println("Ustvarjam trikotnik");
    a = stranA;
    b = stranB;
    c = stranC;
    barva = barvaTrikotnika;
}
```

Ključna beseda `this`

Včasih se metoda sklicuje na predmet, ki jo je sprožil.

- **this** se lahko uporablja znotraj katere koli metode za sklicevanje na trenutno instanco.
- Izogiba se konfliktom v imenskem prostoru — skrivanje spremenljivk instance.

Parametriziran konstruktor z `this`:

```
Trikotnik(int a, int b, int c, String barva){
    System.out.println("Ustvarjam trikotnik");
    this.a = a;
    this.b = b;
    this.c = c;
    this.barva = barva;
}
```

Metode

Metode, definirane v razredu, lahko omogočajo **dostop** do podatkov, delujejo kot **vmesnik** ali pa upravljajo **notranje** stanje predmetov ali samega razreda.

```
public int obseg(){
    return a+b+c;
}
```

Uporaba metode:

```
Trikotnik tri = new Trikotnik(5,6,7,"rdeča");
tri.obseg();
```

Pomen `static`

`static` ustvari člana razreda, ki ga je mogoče uporabiti samostojno, brez sklicevanja na specifično instanco.

Spremenljivke, deklarirane kot `static`, so v bistvu globalne spremenljivke.

- Ko so predmeti razreda instancirani, statične spremenljivke niso kopirane.
- Vse instance razreda si delijo isto statično spremenljivko.

Metode, deklarirane kot `static`, imajo nekaj omejitev:

- Lahko neposredno kličejo samo druge statične metode svojega razreda.
- Lahko neposredno dostopajo samo do statičnih spremenljivk svojega razreda.
- Ne morejo se sklicevati na `this` ali `super`

Uporaba `static`

Primer:
Štetje, koliko trikotnikov je bilo ustvarjenih

```
public class Trikotnik {
    int a,b,c;
    String barva;
    static int stevec = 0;    // statična spremenljivka

    Trikotnik(int a, int b, int c, String barva){
        stevec++;         // povečanje statične spremenljivke
        System.out.println("Ustvarjam trikotnik");
        this.a = a;
        this.b = b;
        this.c = c;
        this.barva = barva;
    }

    public static void main(String[] args) {
        Trikotnik tri1 = new Trikotnik(5,6,7,"rdeča");
        Trikotnik tri2 = new Trikotnik(5,6,7,"zelena");
        Trikotnik tri3 = new Trikotnik(5,6,7,"rumena");
        Trikotnik.stevec;    // rezultat: 3
    }
}
```

`final` lastnosti

Lastnosti deklarirane kot `final` ni moč spreminjati

```
public class Trikotnik {
    int a,b,c;
    final String barva;

    Trikotnik(int a, int b, int c, String barva){
        System.out.println("Ustvarjam trikotnik");
        this.a = a;
        this.b = b;
        this.c = c;
        this.barva = barva;
    }

    public static void main(String[] args) {
        Trikotnik tri = new Trikotnik(5,6,7,"rdeča");
        tri.barva = "oranžna";    // NI DOVOLJENO
    }
}
```

Nadzor dostopa

Ureja dostop delov programa do članov razreda.

S **nadzorom dostopa** lahko preprečimo napačno uporabo razreda.

- Pravilno implementiran razred deluje kot **“črna škatla”**
  - Dobro definiran nabor metod ustvarja vmesnik za uporabo razreda
  - Razred se lahko uporablja, ne da bi razkril svoje notranje delovanje.

Nadzor dostopa

Modifikatorji dostopa nadzorujejo dostop do članov razreda.

Modifikator dostopa	Opis
public	Lahko dostopa katerakoli koda.
private	Dostop imajo samo drugi člani istega razreda.
default	Dostop je omogočen le znotraj istega paketa (package).
protected	Dostop imajo vsi razredi v istem paketu kot nadrazred in vsi podrazredi nadrazreda.

```
public class Triangle {
    private int a,b,c;
    private String color;
    protected static int count = 0;
}
```

Metode - `getters` in `setters`

Getterji: Metode, ki omogočajo zunanji kodi, dostop do vrednosti zasebnih lastnosti razreda.

```
public String getBarva() {
    return barva;
}
```

Setterji: Metode, ki zunanji kodi omogočajo nastavljanje vrednosti zasebnih lastnosti razreda.

```
public void setBarva(String barva) {
    this.barva = barva;
}
```

Življenski cikel predmetov

Vsak predmet ima svoj življenjski cikel:

- rodi se (ga instanciramo)
- lahko ga poizvedujemo ali spreminjamo (ga uporabljamo)
- na koncu umre (ga uničimo)

Alokacija in dealokacija predmetov

- Predmeti so dinamično alocirani z uporabo operatorja **new**.
- V nekaterih jezikih (npr. tradicionalni C++) je treba dinamično alocirane predmete ročno sprostiti z uporabo operatorja **delete**.
- Java uporablja drugačen pristop: **samodejno** upravlja dealokacijo predmetov.
- Ta proces se imenuje **garbage collection**.

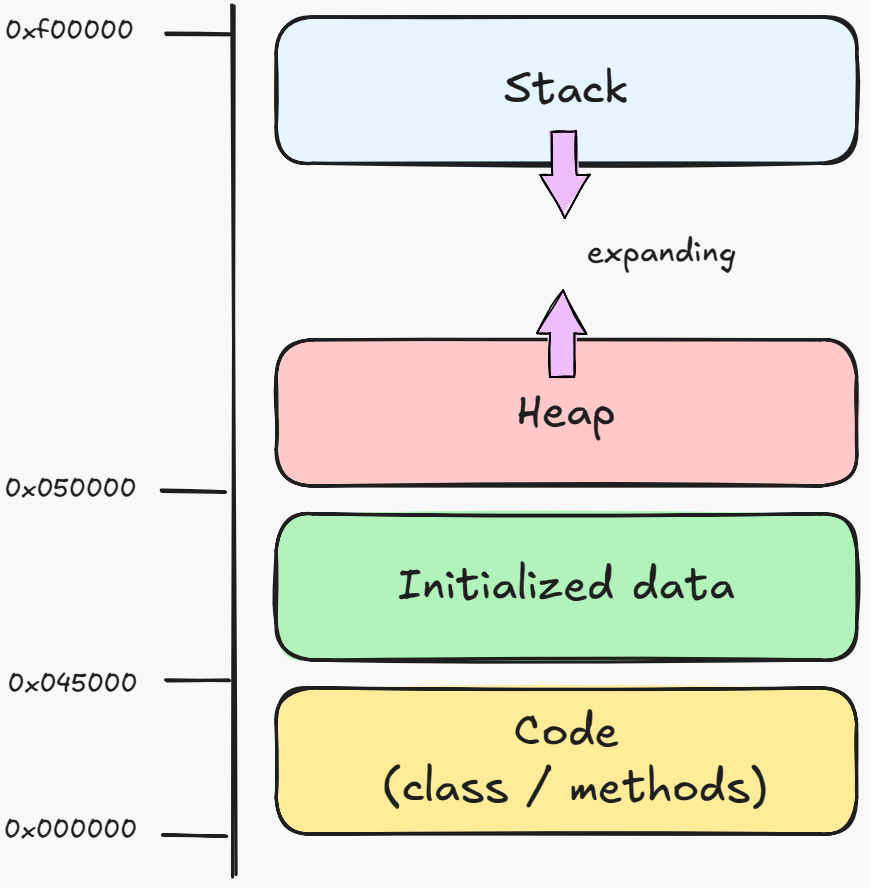
Razporeditev pomnilnika programa

Pomnilnik programa

- Naslovni prostor namenjen programu
- Shranjuje: podatke in ukaze

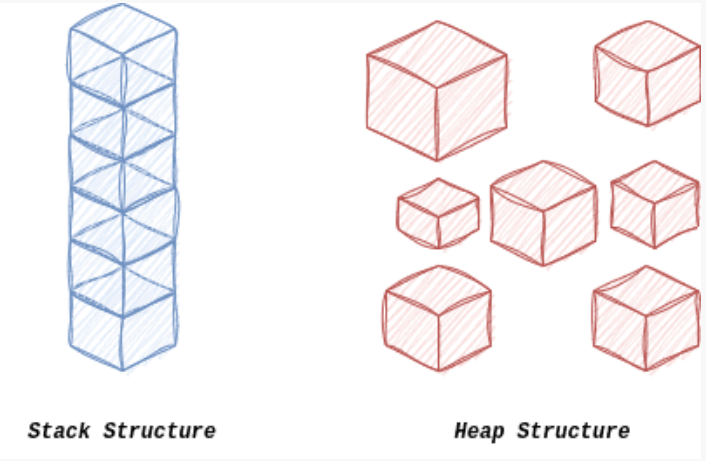
Pomnilnik programa je sestavljen iz:

- Sklad - LIFO podatkovna struktura
  - Uporabljen pri klicu funkcij / metod
  - Samodejna dodelitev - sprostitev
- Heap
  - Hrani življensko daljše strukture (java: objekti)
  - Dinamična dodelitev med izvajanjem



Sklad vs Heap

Sklad (Stack)	Heap
Začasni podatki funkcij	Podatki, ki niso vezani na določeno funkcijo
Podatkovna struktura LIFO	Brez določene strukture
Samodejna dodelitev in sprostitev	Dinamična dodelitev in sprostitev med izvajanjem
Omejena, fiksna velikost	Velikost se dinamično prilagaja
Enostaven za upravljanje	Zahtevnejši za upravljanje
Hiter dostop	Počasnejši dostop
Uporablja se za manjše količine podatkov	Uporablja se za velike in kompleksne strukture podatkov



Življenski cikel predmetov

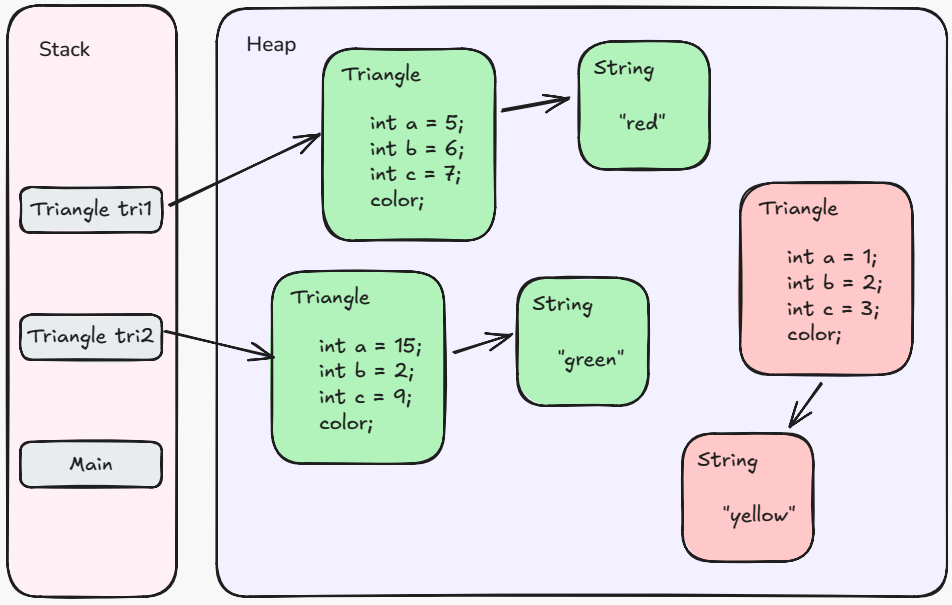
Vsak predmet ima svoj življenjski cikel:

- rodi se (ga instanciramo)
- lahko ga poizvedujemo ali spreminjamo (ga uporabljamo)
- na koncu umre (ga uničimo)

Alokacija in dealokacija predmetov

- Predmeti so dinamično alocirani z uporabo operatorja **new**.
- V nekaterih jezikih (npr. tradicionalni C++) je treba dinamično alocirane predmete ročno sprostiti z uporabo operatorja **delete**.
- Java uporablja drugačen pristop: **samodejno** upravlja dealokacijo predmetov.
- Ta proces se imenuje **garbage collection**.

Garbage collection v Javi



Garbage collection deluje tako, da sprosti pomnilnik, ko **ni več nobenih referenc na predmet**.

Predmeti v **rdečem** bodo dealocirani

- Ni več reference do njih

- Če predmet ni več potreben, se zaseden pomnilnik lahko samodejno sprosti.
- Ni potrebe (in tudi ni načina v Javi) za eksplicitno uničenje predmetov.
- Garbage collection se dogaja občasno
  - Lahko se zgodi kadarkoli med izvajanjem
  - Lahko povzroči težave ...

Podajanje argumentov

Obstajata dva načina, kako lahko programski jezik posreduje argument funkciji.

Prenos po vrednosti:

- Vrednost argumenta se kopira v parameter
- Spremembe parametra nimajo vpliva na argument.

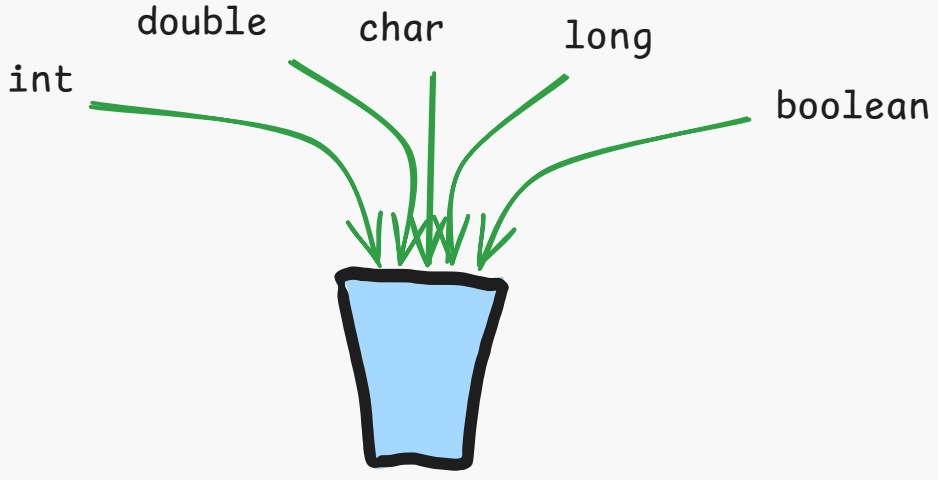
Prenos po referenci:

- Referenca na argument se posreduje parametru
- Referenca se uporablja za dostop do dejanskega argumenta, ki je podan pri klicu
- Spremembe parametra bodo vplivale na argument

Podajanje argumentov - Prenos po vrednosti

Spremenljivke so kot skodelice fiksne velikosti.

Primitivni tipi v Javi so dovolj majhni, da preprosto stojijo v skodelici.



Podajanje argumentov - Prenos po vrednosti

Primer:

```
public static void main(String[] args) {
    int x = 12;
    incPrint(x);
    System.out.println(x);
}

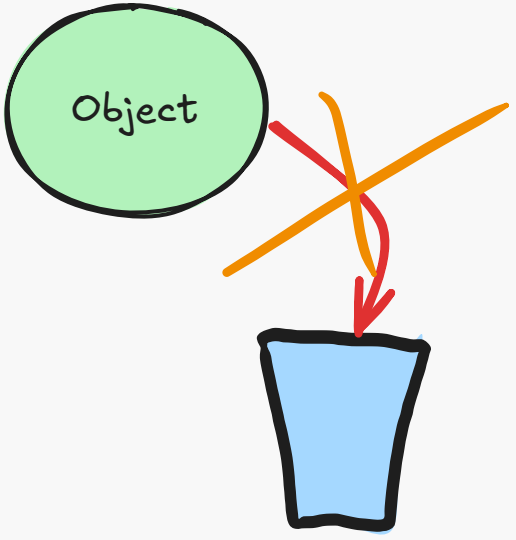
public static void incPrint(int x){
    ++x;
    System.out.println(x);
}

izpis:
13
12
```

Podajanje argumentov - Prenos po referenci

Predmeti so preveliki, da bi jih shranili v spremenljivko

- Shranjeni so nekje drugje
- Spremenljivka shrani številko (**referenco**), ki locira predmet



Podajanje argumentov - Prenos po referenci

Primer:

```
public static void main(String[] args) {
    int[] x = {1};
    incPrint(x);
    System.out.println(x[0]);
}

public static void incPrint(int[] x){
    x[0] = 1000;
    System.out.println(x[0]);
}

output:
1000
1000
```

Podajanje argumentov - Prenos po referenci

Primer: podajanje in vračanje predmetov

```
public Trikotnik vrniVečji(Trikotnik drugi){
    if( drugi.obseg() > this.obseg() ){
        return drugi;
    }
    return this;
}

public static void main(String[] args) {
    Trikotnik tri1 = new Trikotnik(5,6,7,"rdeča");
    Trikotnik tri2 = new Trikotnik(11,12,13,"rumena");
    Trikotnik večji = tri1.vrniVečji(tri2);
    System.out.println(večji.barva);
}

izpis:
rumena
```

Sedaj imava dve referenci, ki kažeta na isto instanco predmet.

Funkcije, vmesniki in pogodbe

Funkcijo si lahko predstavljamo kot **črno škatlo**, ki za nas nekaj naredi:

- **Uporabniški vmesnik** – API
- **Primer**: X(u, v) → z

Ključna točka:

- Ni nam pomembno **kako** je nekaj narejeno, pomembno je samo **kaj** je narejeno!

Ta pogled omogoča neodvisno delo v ekipi.

Za uporaben opis funkcije potrebujemo:

1. Opis, kako klicati funkcijo (**podpis**)
2. Opis, kaj funkcija dejansko naredi (**pogodba**)

Podpis funkcije

Primer funkcije:

```
int vsotaDo0(int x) {
    int vsota = 0;
    while (x > 0) {
        vsota += x;
        x--;
    }
    return vsota;
}
```

Podpis funkcije:

```
int vsotaDo0(int x)
```

- **Povratni tip**: `int`
- **Ime funkcije**: `vsotaDo0`
- **Parametri**: `int x` (tip in ime parametra)

Pogodbe

Pogodba veže tistega, ki funkcijo **uporablja** in tistega, ki funkcijo napiše (**implementira**)

- prvi ve, kaj lahko pričakuje in zahteva od funkcije.
- drugi ve, kaj je mora narediti.

Formalna pogodba se lahko uporablja za **formalno** dokazovanje obnašanja programa.

- Za naše pogodbe bomo uporabljali (pol-)formalno notacijo.

Členi pogodbe

Opis:

```
Desc:
    Funkcija izračuna vsoto števil od 0 do x.
```

Parameteri:

```
Params:
    x – operand [in]
```

Predpogoj:

```
Pre: x > 0
```

Členi pogodbe

Popogoj:

```
Post: RESULT > 0
```

Rezultat:

```
Result: x + (x-1) + (x-2) + ... + 1
```

Okolje:

```
Env:    -
```

Primer pogodbe

```
/*
 * Desc: Funkcija izračuna vsoto števil od 0 do x.
 * Params: x – operand [in]
 * Pre: x > 0
 * Post: RESULT > 0
 * Result: x + (x-1) + (x-2) + ... + 1
 * Env:  -
 *
 */
public int vsotaDo0(int x) { . . . }
```