Integrirana razvojna okolja

Delo včasih

1. Urejanje kode z urejevalniki besedil (VIM, Emacs, beležnica, Notepad++...).

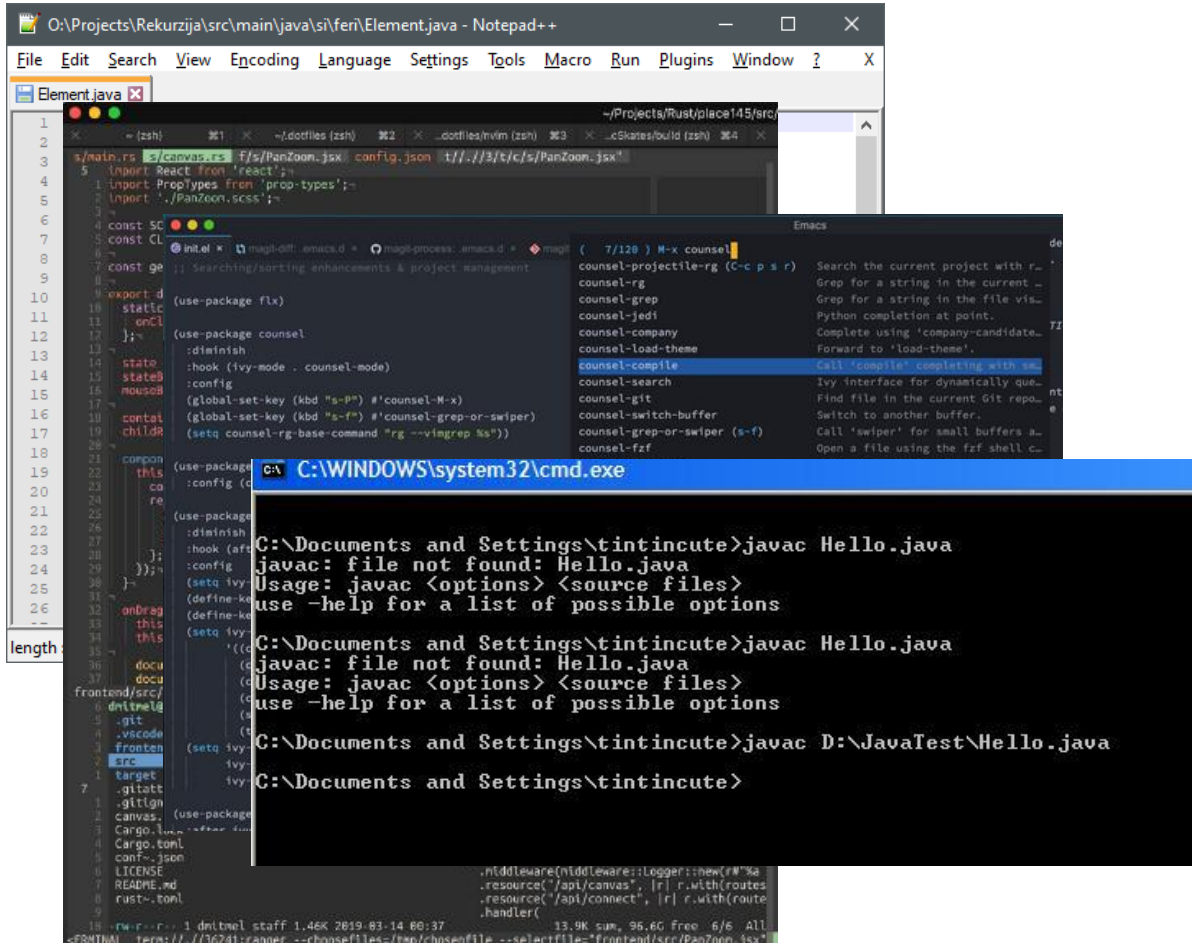
2. Prevajanje, povezovanje in poganjanje kode v ukazni vrstici.

3. Razhroščevanje v ukazni vrstici.

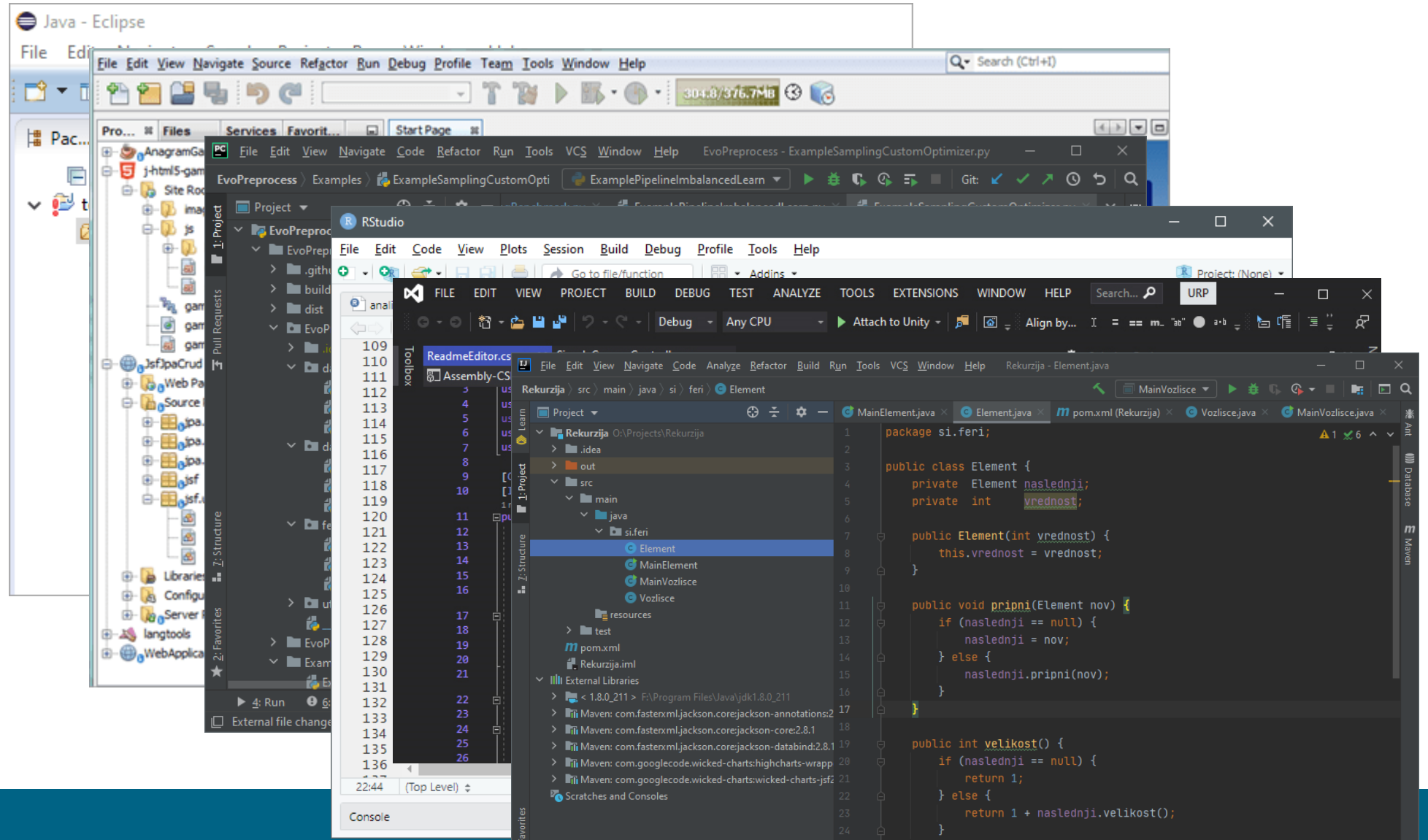
ali še hujše

3. “Razhroščevanje” z ukazi izpisa.

```
System.out.println(oseba.ime);
console.log(oseba.ime);
print(oseba.ime)
```



Delo danes

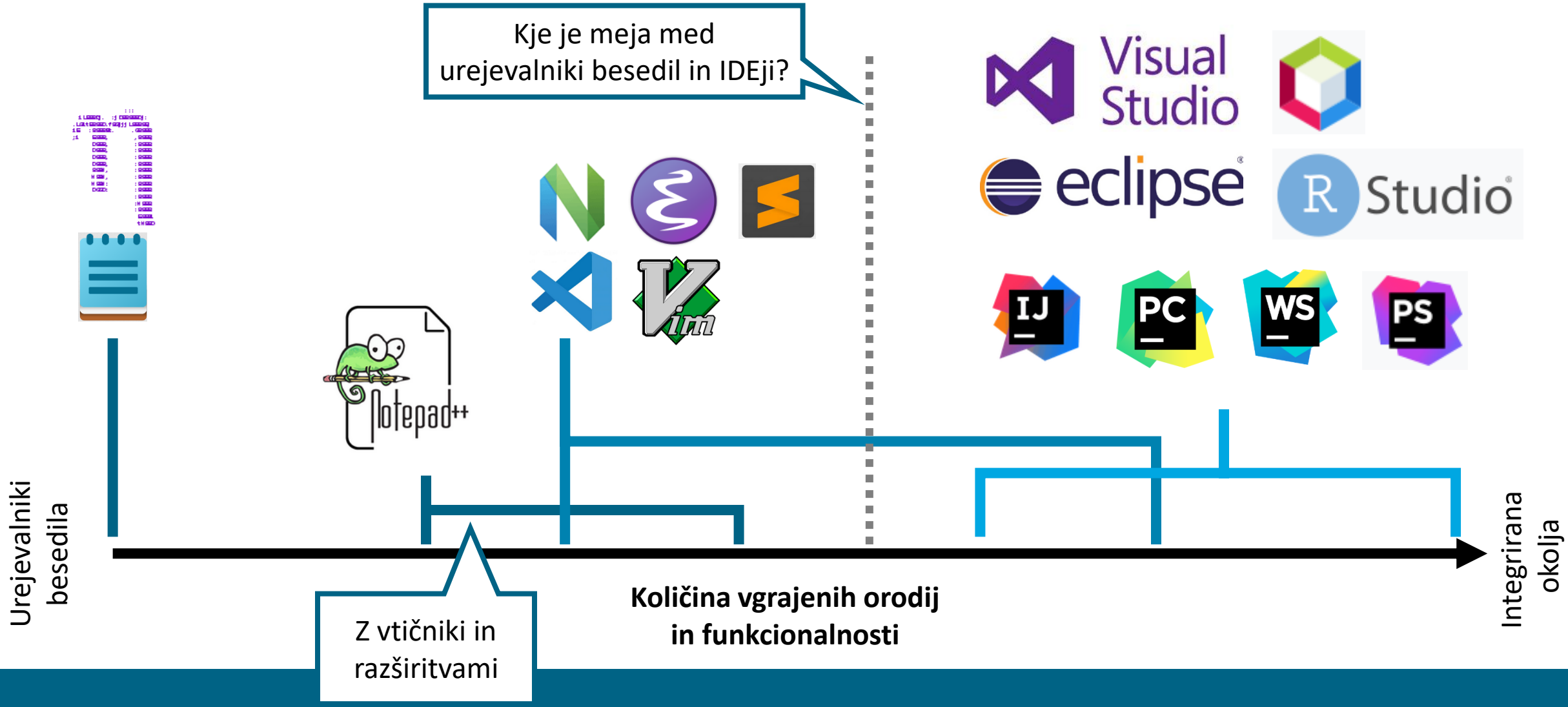

Java - Eclipse
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
File Edit View Navigate Code Refactor Run Tools VCS Window Help
EvoPreprocess - ExampleSamplingCustomOptimizer.py
RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
FILE EDIT VIEW PROJECT BUILD DEBUG TEST ANALYZE TOOLS EXTENSIONS WINDOW HELP
Rekurzija - Element.java
package si.feri;
public class Element {
private Element naslednji;
private int vrednost;
public Element(int vrednost) {
this.vrednost = vrednost;
}
public void pripni(Element nov) {
if (naslednji == null) {
naslednji = nov;
} else {
naslednji.pripni(nov);
}
}
public int velikost() {
if (naslednji == null) {
return 1;
} else {
return 1 + naslednji.velikost();
}

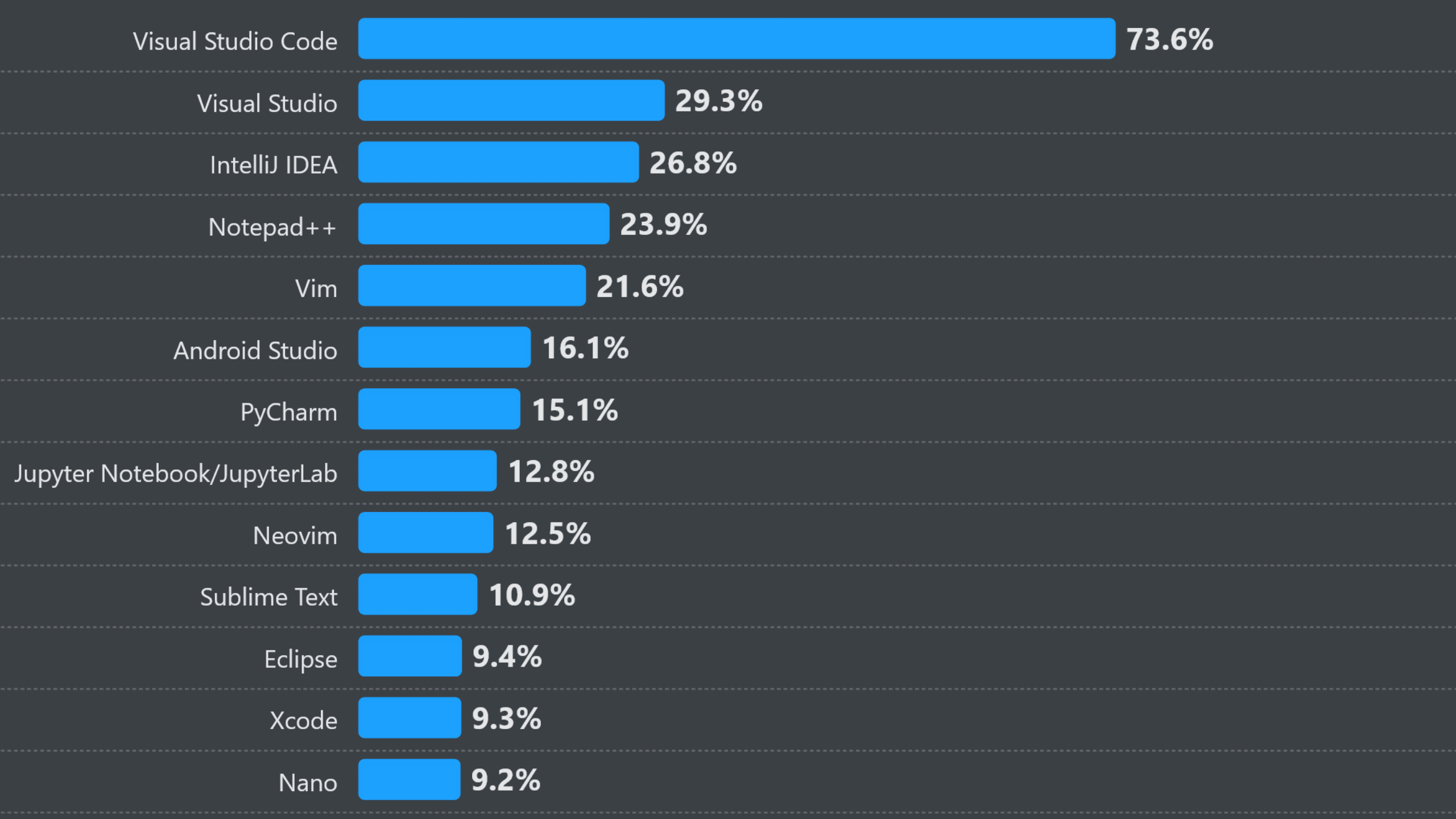
Integrirano razvojno okolje

- **Integrirano razvojno okolje** (angl. *integrated development environment*) ali **IDE** je program, ki nudi funkcionalnosti in povezuje nabor **orodij** za razvoj programske opreme ali drugih sistemov, ki se razvijajo z računalnikom.
  - Opravijo precej rutinskega dela (npr. sprotno preverjanje sintakse, dopolnjevanje kode, razhroščevanje, omogočajo sodobne pristope...).
  - Omogočajo, da se razvijalci bolj posvetijo problem in ne pa samemu procesu razvoja.

- **Prednost**: večja produktivnost razvijalcev.
- **Slabost**: zaradi kompleksnosti zahtevajo čas za učenje in privajanje, preden lahko izkoristimo del njihovih funkcionalnosti za povečanje produktivnosti.

IDE vs urejevalniki besedila




Visual Studio Code
73.6%
Visual Studio
29.3%
IntelliJ IDEA
26.8%
Notepad++
23.9%
Vim
21.6%
Android Studio
16.1%
PyCharm
15.1%
Jupyter Notebook/JupyterLab
12.8%
Neovim
12.5%
Sublime Text
10.9%
Eclipse
9.4%
Xcode
9.3%
Nano
9.2%

Orodja za razvoj programske opreme

Orodja za razvoj programske opreme

- **Upravljanje s projektom**
  - Upravljanje z verzijami datotek (P1) ✓
  - Orodja za avtomatizacijo postopkov (RIS) ✓
  - Upravljanje s paketi ←
- **Zagon programske opreme**
  - Prevajalnik ✓
  - Interpreter ✓
  - Optimizator ←
  - Profiler ←
  - Razhroščevalnik ←
- **Urejevalnik izvorne kode**
  - Sprotno preverjanje sintakse ✓
  - Preoblikovanje ←
  - AI pomoč pisanja kode !
- **Analiza kode**
  - Linter ←
  - Statični in dinamični analizatorji !
- **Podpora za objektno ali funkcijsko programiranje**
  - Class browser, object inspector, class hierarchy diagram !
  - Method browser !
- **Načrtovanje sistemov**
  - Izdelava grafičnih vmesnikov (GUI) ✓
  - Diagramske tehnike: UML, ER… !
- **Delo s podatki**
  - Podatkovne baze (povezava, urejanje) ✓
  - Delo z CSV, JSON, XML datotekami… !

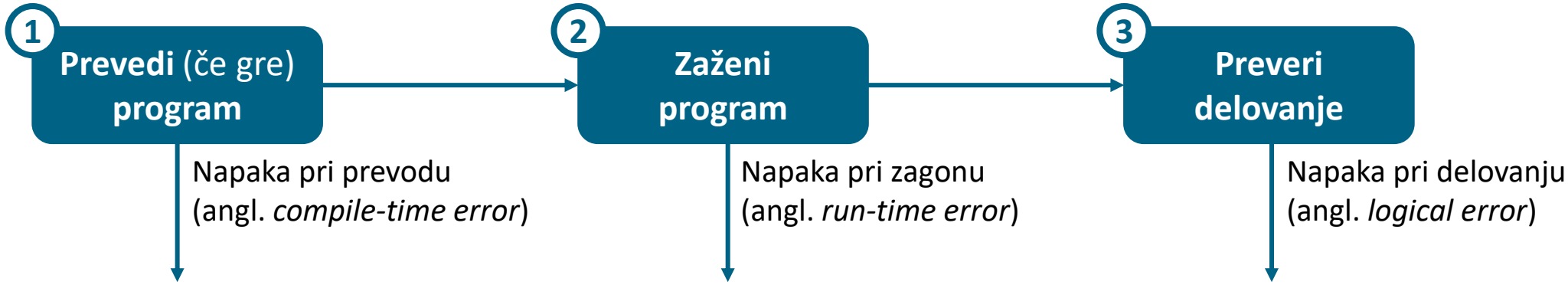
Razhroščevanje

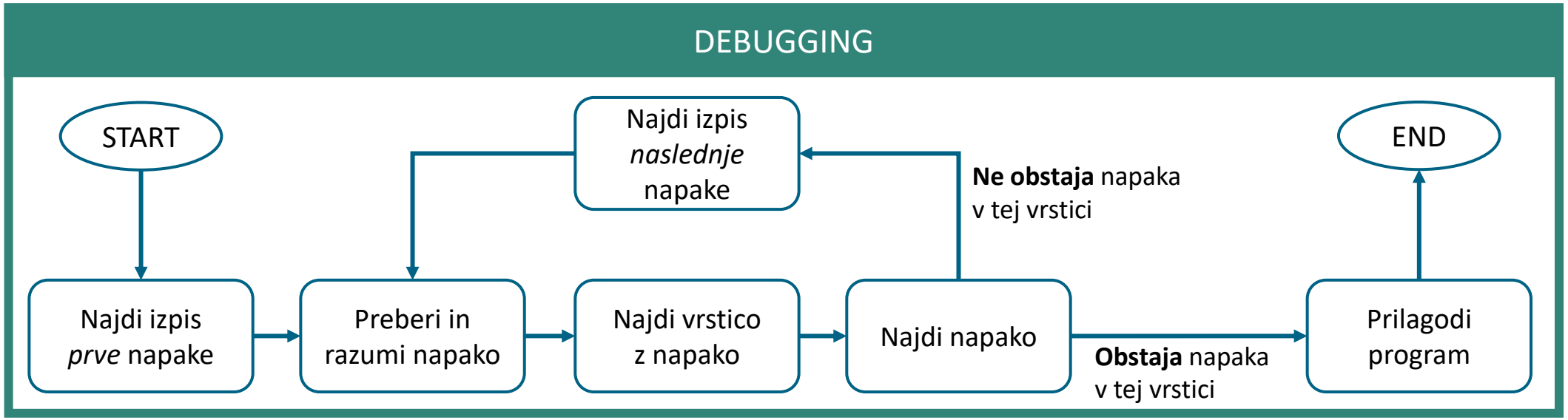
Osnovna terminologija

- **Odpoved** (angl. *failure*): Vsak odklon opazovanega obnašanja od specificiranega obnašanja.
- **Zanesljivost** (angl. *reliability*): Merilo ujemanja obnašanja sistema s specifikacijami obnašanja.

- **Napaka** (angl. *error*): Sistem je v takšnem stanju, da ga bo nadaljnje izvajanje privedlo do odpovedi.
- **Hrošč** (angl. *bug*): Vzrok za napako.
- **Razhroščevanje** (angl. *debugging*): Postopek iskanja in odpravljanja napak.

Sistematično razhroščevanje

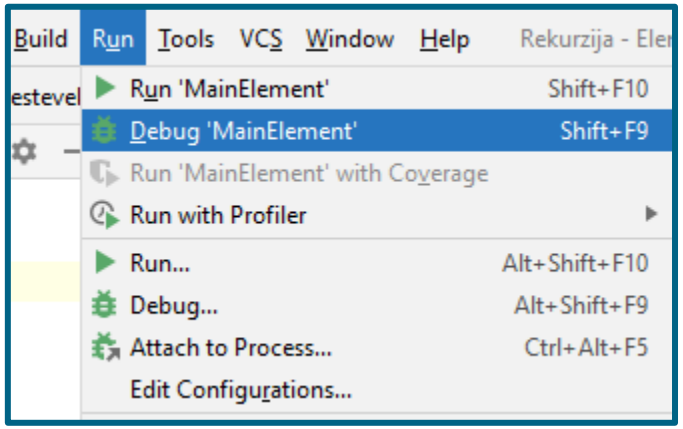




Razhroščevalnik

- **Razhroščevalnik** (angl. *debugger*) je program, ki zaganja kodo v kontroliranem okolju za namen iskanja hroščev in odpravljanja napak.

- Pri zagonu programa z razhroščevalnikom lahko pregledujemo shranjene vrednosti programa in zaganjamo kodo.
  - Zagon v načinu razhroščevanje zahteva poseben zagon, ki je počasnejši od navadnega zagona programa.

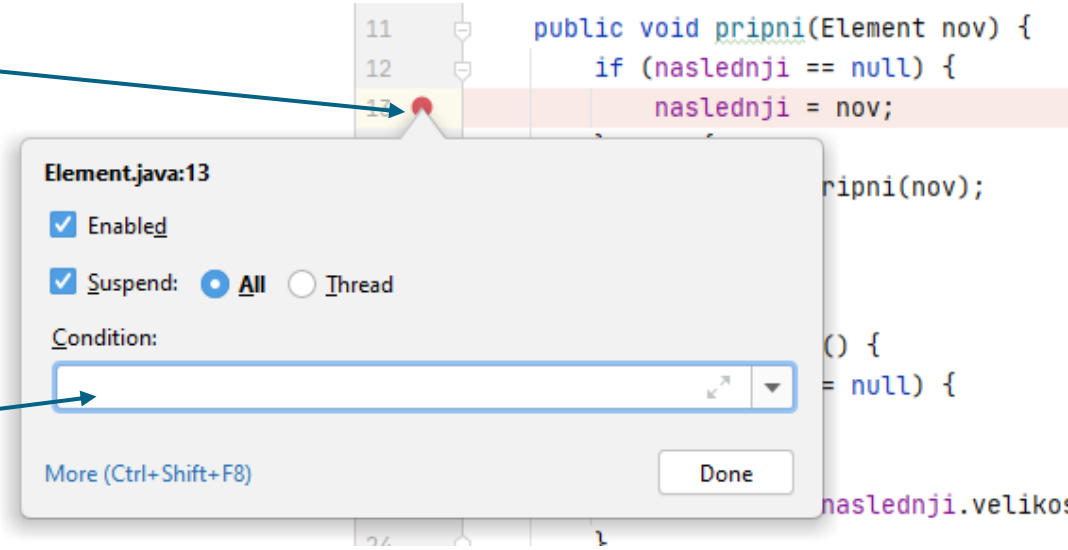


- Zagon programa se ustavi in nam prepusti kontrolo ko:
  a) pride do **napake** pri zagonu, ali
  b) naleti na vnaprej postavljeno **točko prekinitve**.

Točka prekinitve

- **Točka prekinitve** (angl. *breakpoint*) je mesto v programski kodi, ki jo določi razvijalec, kjer želi, da se zagon programa ustavi.

- V kodi izberemo kdaj želimo, da se zagon programa prekine in mi (razvijalci) prevzamemo kontrolo.
  - Po navadi se določi pred ali v problematičnem delu.
  - Od tam naprej se delovanje spremlja vsak korak.
- Nekateri razhroščevalniki omogočajo dodajo pogojev prekinitve.




```
oseba == null
oseba.getIme().equals('Janez')
```

Pišemo v jeziku v katerem programiramo

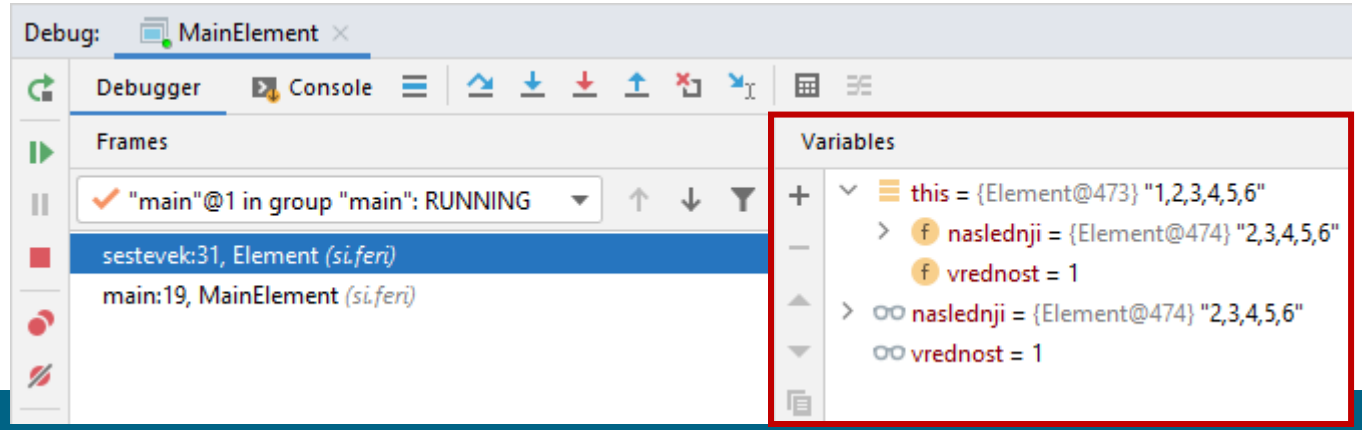
Stanje prekinitve

- Nekateri IDE-ji vrednosti spremenljivk kažejo med kodo ali kot tooltip, ko z miško gremo na spremenljivko.

```
27      public int sestevek() {
28          if (naslednji == null) {
29              return vrednost;
30          } else {
31              return vrednost + naslednji.sestevek();  vrednost: 1  naslednji: "2,3,4,5,6"
32          }
33      }
```

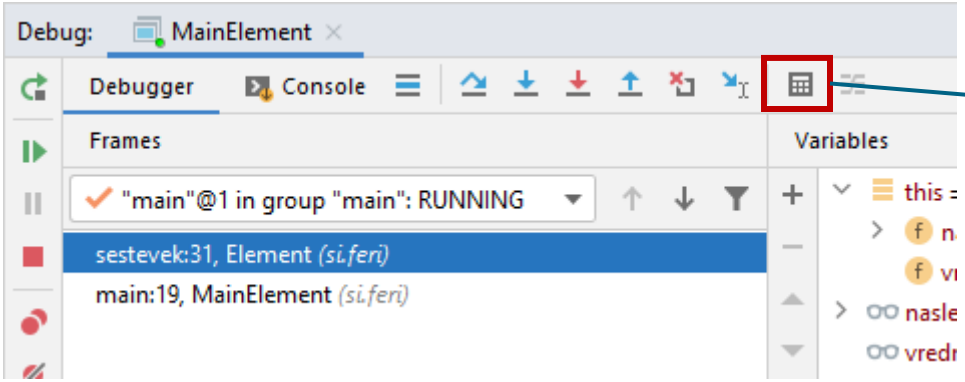
1

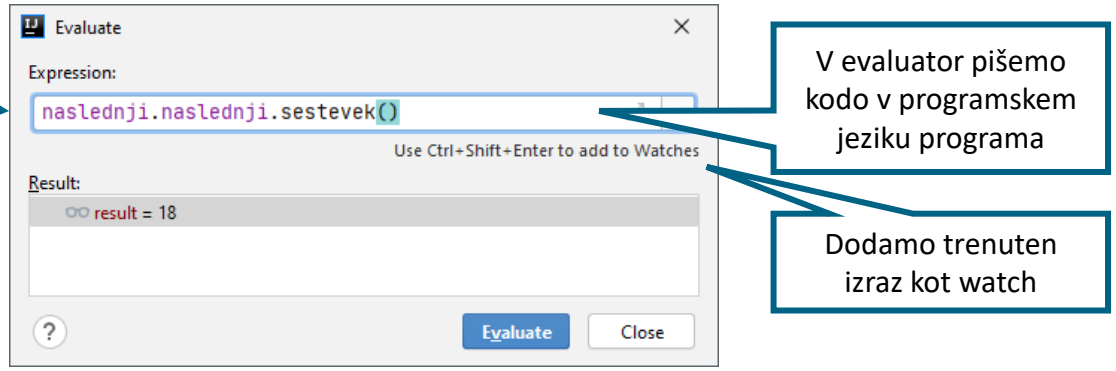
- Vso trenutno stanje spremenljivk je prikazano v za to namenjenem delu.



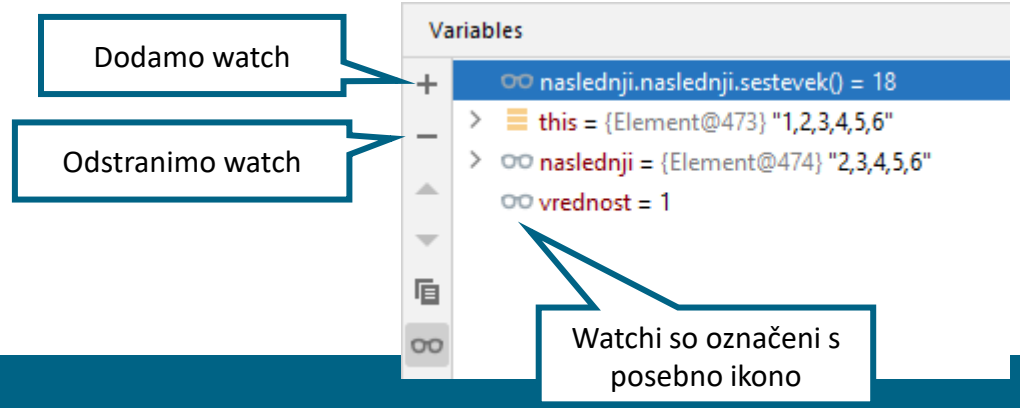
Stanje prekinitve

- Tudi ročno lahko pregledamo stanje spremenljivk, tako da zaženemo evalvator.



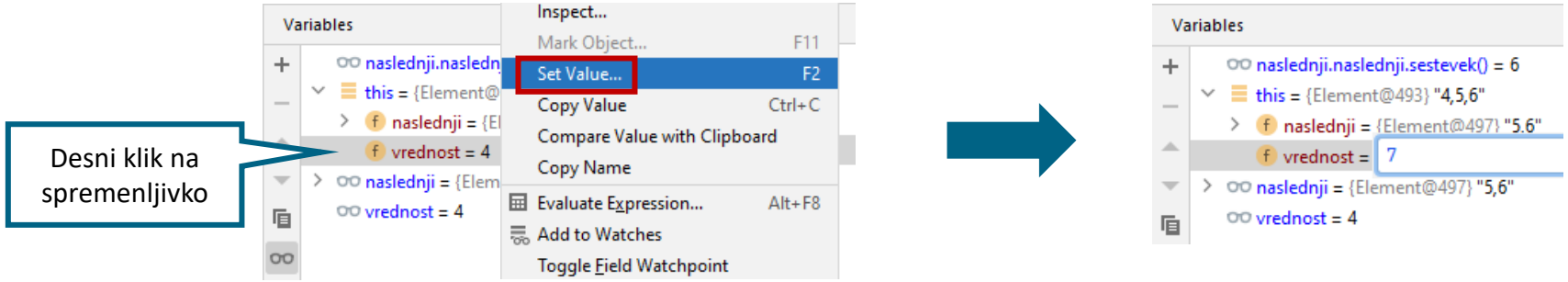


- Izraze iz evalvatorja lahko permanentno shranimo v pregled – dodamo **watch**.

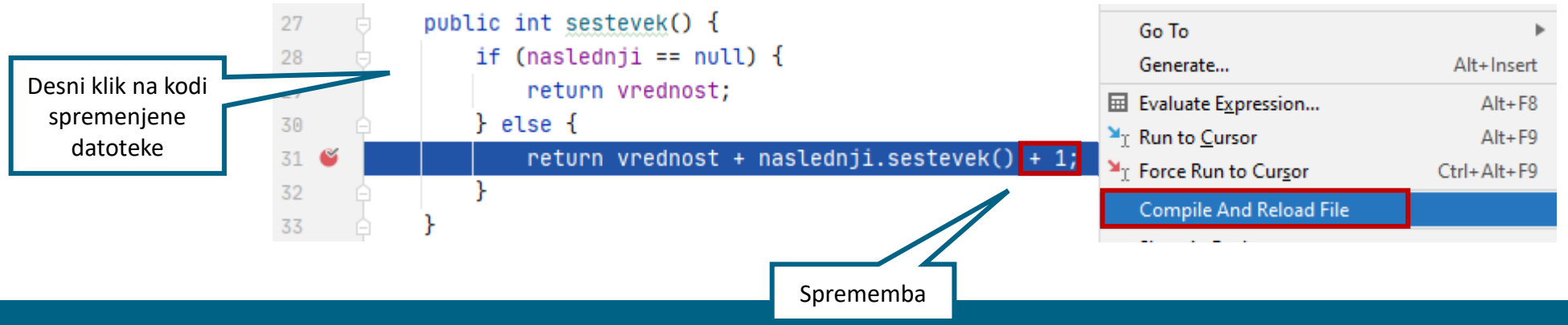


Spreminjanje stanja prekinitve

- Med prekinitvijo lahko spremenimo vrednosti spremenljivk.

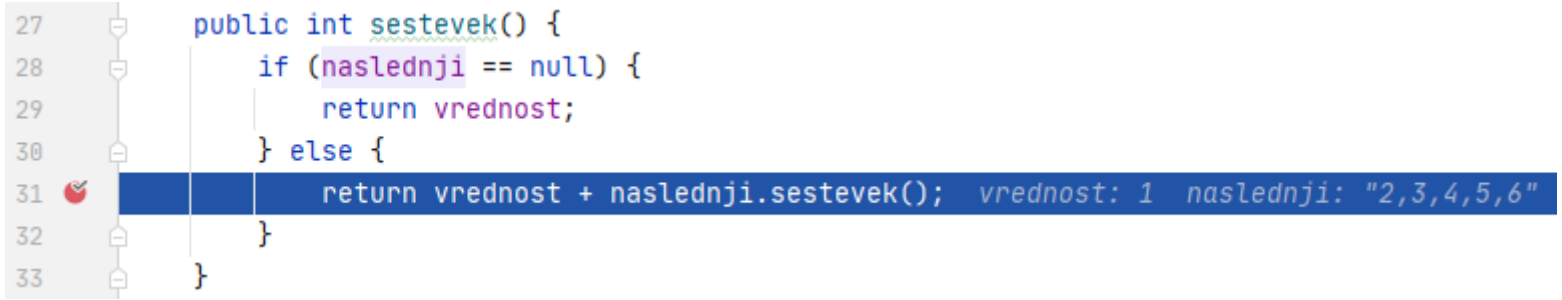


- Lahko pa spremenimo kar izvorno kodo in nadaljujemo z izvajanjem na novi verziji, brez ustavitve izvajanja programa.



Kontroliranje prekinitve

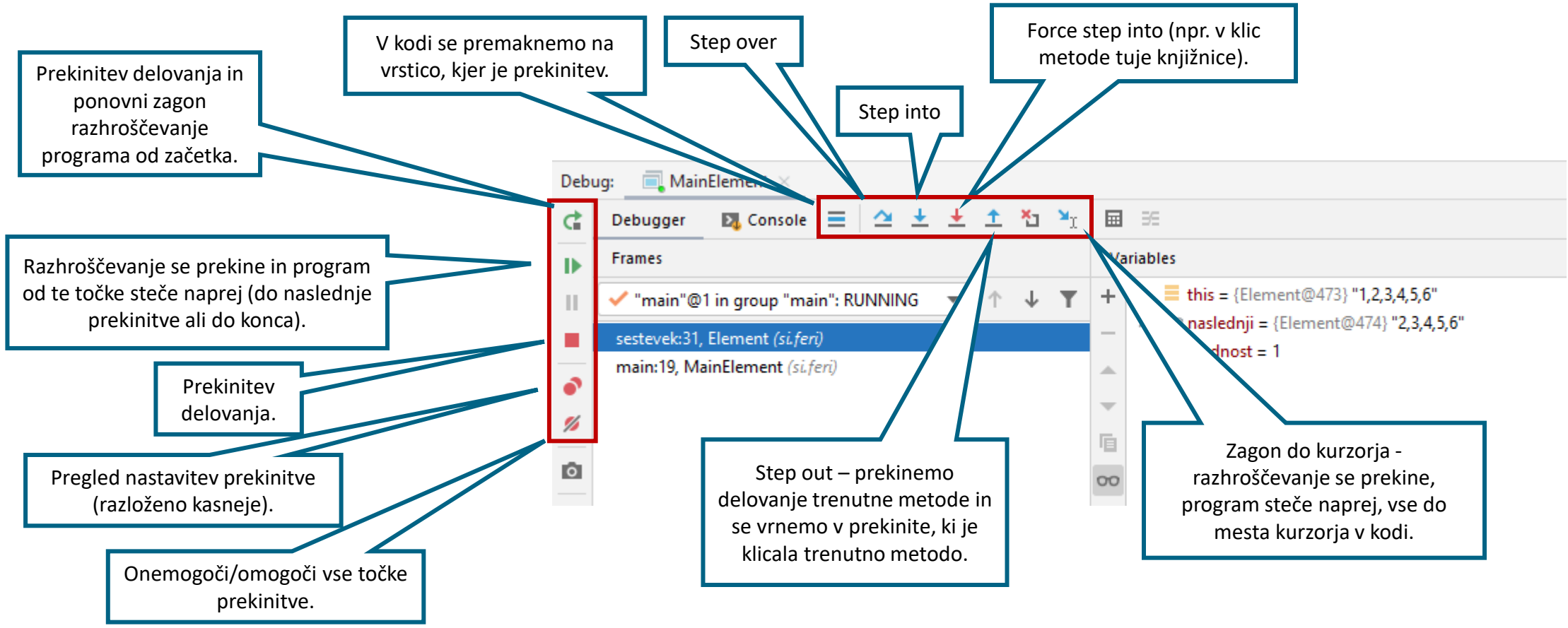
- V delu IDE-ja še vedno vidimo programsko kodo, kjer je označena vrstica, ki še ni bila zagnana, je pa naslednja v vrsti.



- Vrstico lahko ročno zaženemo na več načinov.
  - Kar zaženemo vrstico in gremo na naslednjo – **preskočimo vrstico** (angl. *step over*). V zgornjem primeru bi se kar `return` izvršil in bi šli na vrstico, ki je klicala metodo `sestevek()`.
  - Gremo vsak del te vrstice naprej razhroščevat (vstopimo v kakšen klic metode, če je v tej vrstici) – **skočimo v vrstico** (angl. *step into*). V zgornjem primeru bi vstopili v prvo vrstico metode `sestevek()` instance `naslednji`.

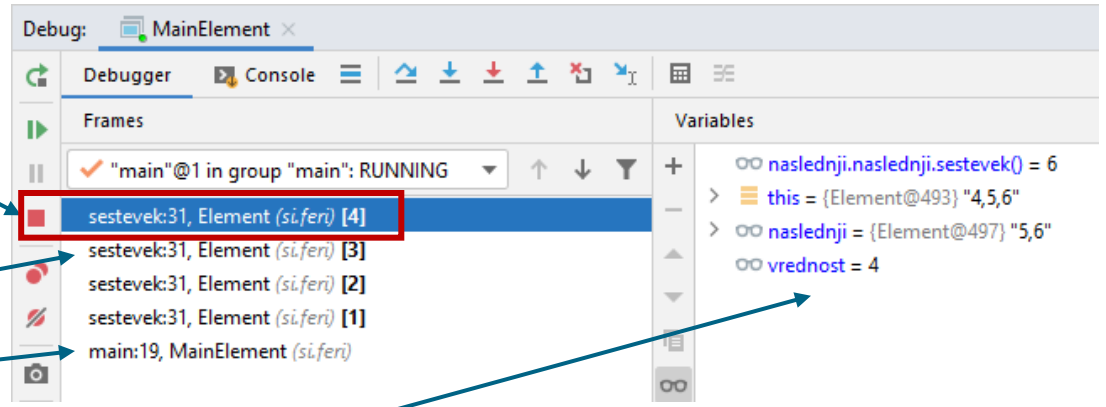
Kontroliranje prekinitve

- S prekinitvijo operiramo v delu IDE-ja, ki je namenjen razhroščevanje.



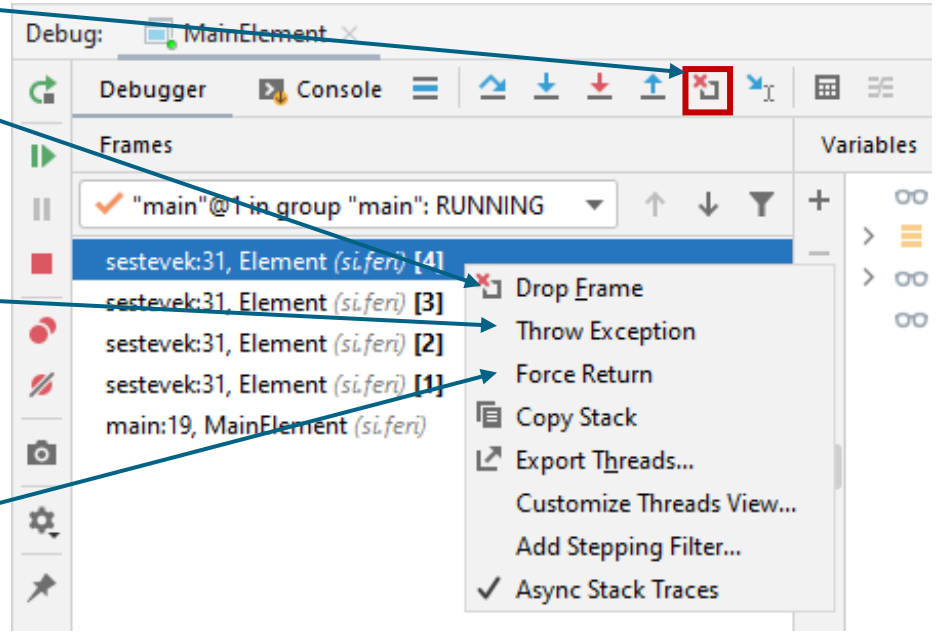
Upravljanje med prekinitvami

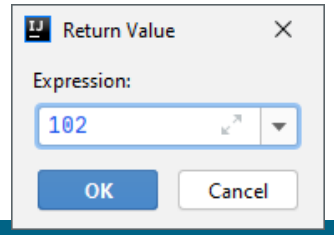
- Metoda, kjer je trenutno točka izvajanja se imenuje **okvir prekinitve** (angl. *frame*).
  - Trenutno se nahajamo v 31. vrstici metode `sestevek()`, ki je v instanci razreda `Element`.
  - To metodo je klicala prav tako metoda `sestevek()` v razredu `Element`...
    - Prvi klic pa je izvršila 19. vrstica v metodi `main()` v instance razreda `MainElement`.
  - Stanje spremenljivk je prikazano le za trenuten okvir.



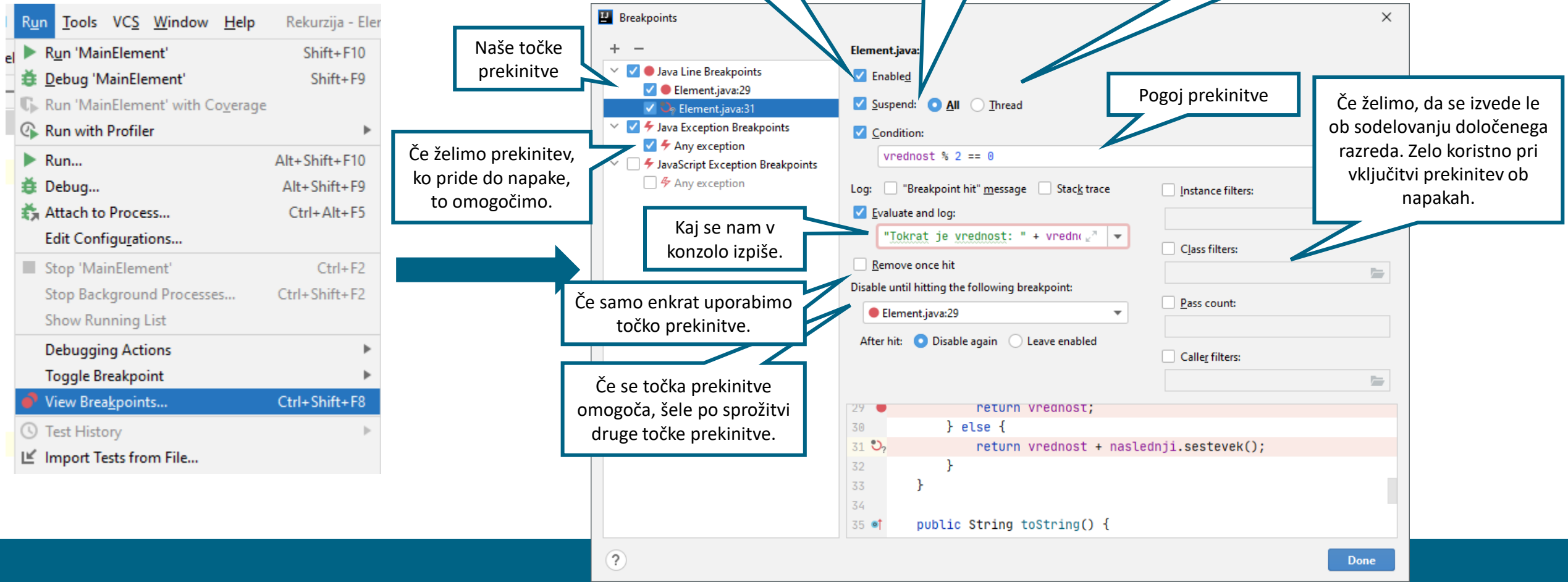
Upravljanje med prekinitvami

- Okvir lahko zavržemo in se vrnemo na prejšnji okvir v seznamu.
- Prekinemo delovanje okvirja z napako. Delovanje se vrne na prejšnji okvir z napako.
- Prekinemo delovanje okvirja tako, da specificiramo kaj se vrne v prejšnji okvir.





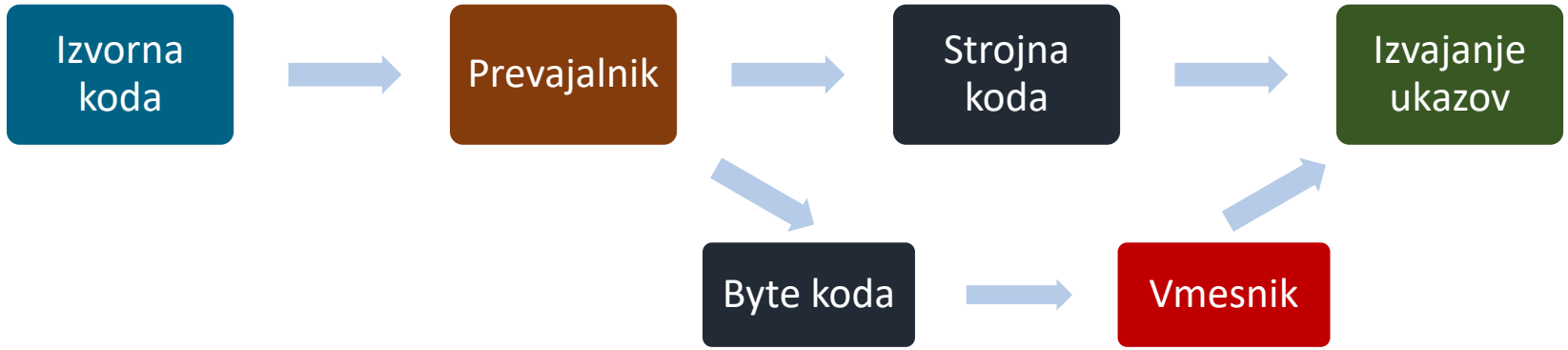
Upravljanje s točkami prekinitve



Orodja za razvoj programske opreme

Zagon programske kode – ponovitev

- **Prevajalnik** (angl. *compiler*) pa je program, ki "prevede" kodo programskega jezika v kodo, ki jo razume (jo lahko zažene) računalnik (npr. strojno kodo). To je proces **prevajanja** (angl. *compiling*).
  - C, C++, Java



- Prevaja lahko direktno v strojno kodo (ki teče točno na določenem OS in točno na določeni strojni opremi) ali v byte-kodo, ki teče na vmesniku imenovanem izvajalno okolje (angl. *runtime environment*).
  - Primeri izvajalnih okolij: JVM za Javo, Kotlin, Scalo in CLR za C#, VB, F#...

Zagon programske kode – ponovitev

- **Tolmač/interpreter** (angl. *interpreter*) je program, ki ukaze programske kode kar sam izvaja na računalniku, brez da kodo prevede v izvedljiv program. To je proces tolmačenja/interpretiranja (angl. *interpreting*).
  - Python, R, Ruby, JavaScript

- Kaj je interpreter JavaScripta?
  - V brskalniku je to JS pogon: V8 v Chrome, SpiderMonkey v Firefoxu…

Zagon programske kode
Reševanje nekompatibilnost

- Novejše verzije JS so kompatibilne z novimi brskalniki, ne pa s starejšimi. To rešujemo s **transpilacijo** (angl. *transpiling*).

- Transpilacija je proces, kjer “prevedemo” kodo napisano v enem izmed programskih jezikov v jezik na istem abstraktnem nivoju.
  - Primer: iz Jave v C#, iz C v JavaScript, iz Java 12 v Java 8.

- Ta proces prevajanja izvede program **transpilator**, včasih transprevajalnik (angl. *transpiler* ali *transcompiler*).

Zagon JavaScript kode

Babel.js

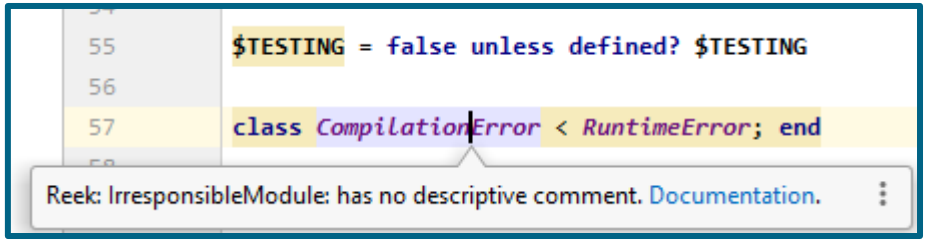
- **Transpilator**, ki prevede novejše verzije JS v starejše verzije.
- S tem omogoči podporo novejšim funkcionalnostim na starejših brskalnikih.
  - Omogoči podporo za JSX, arrow funkcije…
  - Primer

Node.js

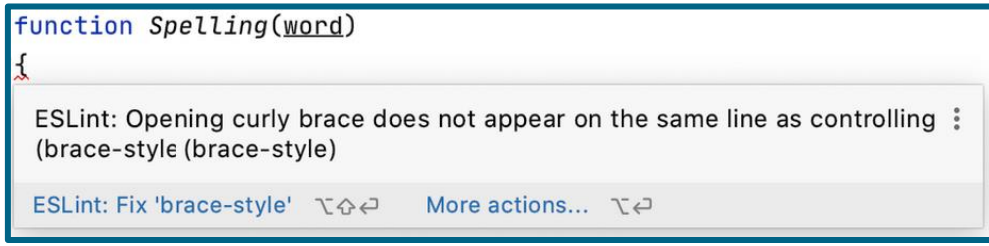
- Je **izvajalno okolje**, ki za izvaja ukaze JS izven brskalniku in pri tem uporablja interpreter V8 za izvajanje ukazov in optimizator kode.
  - Optimizator našo JS kodo nadomesti z bolj optimalno in v C napisano kodo.
- Mi smo že uporabljalo Node.js program strežnika za izvajanja zaledja naše spletne aplikacije.

Analiza izvorne kode
Linter

- **Linter** je orodje, ki **analizira** izvorno kodo in **označuje** napake, hrošče, slogovne pomanjkljivosti in potencialno slabo kodo (npr. *code smell*) ter s tem izboljšuje kakovost in vzdržljivost kode.

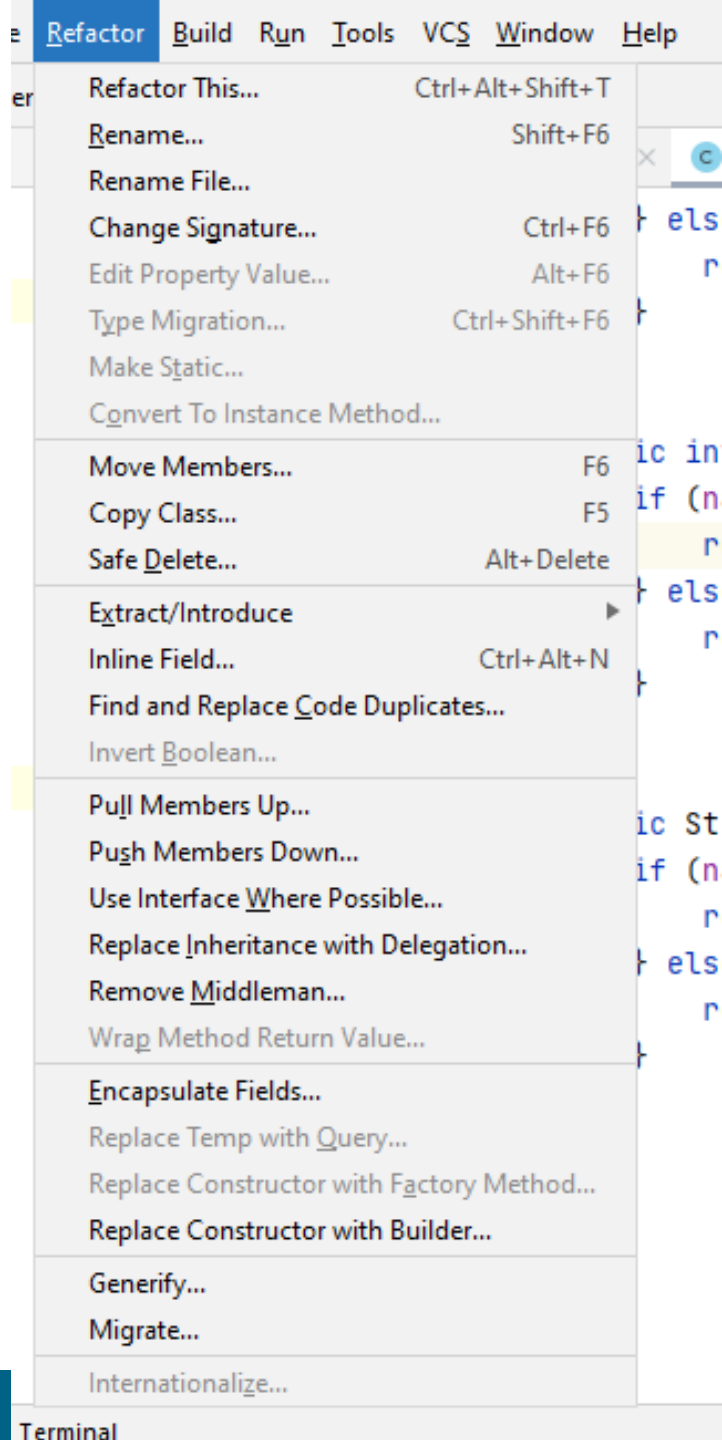


- Funkcionalnosti:
  - Odkrivanje sintaktičnih napak
  - Priporočanja standardov kodiranja in najboljših praks
  - Ugotavljanje morebitnih varnostnih ranljivosti ali težav z delovanjem
  - Poudarjanje neuporabljenih spremenljivk in uvozov
  - Predlaganje alternativnih načinov pisanja kode, ki so lahko učinkovitejši ali bolj berljivi
  - Olajšanje sodelovanja z zagotavljanjem doslednosti in enotnosti sloga in strukture kode



Urejanje izvorne kode
Preoblikovanje

- **Preoblikovanje** (angl. *refactoring*) kode je proces, ki izboljša notranjo strukturo kode (izgled kode) brez spremembe delovanja in funkcionalnosti kode.
- Razlogi za preoblikovanje:
  - neprimerno in nekonsistentno poimenovanje,
  - podvojena koda,
  - predolga metoda ali prevelik razred,
  - odvečne spremenljivke,
  - predolg nabor parametrov…



Urejanje izvorne kode
Tehnike preoblikovanja

- Za boljšo razumljivost in sledenje standardom poimenovanja in strukture:
  - **Preimenovanje** – med razvojem lahko razredi in metode dobijo drugačen pomen ali funkcionalnost.
  - **Premik metode** – metodo premaknemo v podrazred ali nadrazred.

- Za manjšo sklopljenost kode in lažjo ponovno uporabo:
  - **Izločitev metode** – del kode prenesemo v novo metodo.
  - **Izločitev razreda** – del kode prenesemo v nov razred.
  - **Generalizacija** – namesto podrazredov uporabimo bolj splošne razrede in s tem izkoristimo prednosti objektno orientiranega programiranja.

- Za boljšo varnost (onemogočanje nezaželenih ukazov):
  - **Enkapsulacija** – spremenljivka postane privatna, kreirajo se metode za dostop.

Primer izločitve metode

```
// Izračun skupne cene izdelkov v nakupovalnem vozičku,
// vključno z davkom
function izracunajSkupnoCeno(izdelki) {
    let skupnaCena = 0;
    let davek = 0.2; // 20% davek
    for (let i = 0; i < izdelki.length; i++) {
        skupnaCena += izdelki[i].cena * izdelki[i].kolicina;
    }
    skupnaCena += skupnaCena * davek;
    return skupnaCena;
}

// Primer izdelkov
const izdelki = [
    {ime: "Kruh", cena: 1.2, kolicina: 2},
    {ime: "Mleko", cena: 0.8, kolicina: 1},
    {ime: "Jabolka", cena: 0.5, kolicina: 5}
];

console.log(izracunajSkupnoCeno(izdelki)); // Izpis skupne cene
```

```
// Izračun cene posameznega izdelka (cena * količina)
function izracunajCenoIzdelka(izdelek) {
    return izdelek.cena * izdelek.kolicina;
}

// Izračun davka na skupno ceno
function dodajDavekNaCeno(cena, davekStopnja) {
    return cena * davekStopnja;
}

// Izračun skupne cene izdelkov v nakupovalnem vozičku, vključno z davkom
function izracunajSkupnoCeno(izdelki) {
    let skupnaCena = 0;
    let davek = 0.2; // 20% davek
    for (let i = 0; i < izdelki.length; i++) {
        skupnaCena += izracunajCenoIzdelka(izdelki[i]);
    }
    skupnaCena += dodajDavekNaCeno(skupnaCena, davek);
    return skupnaCena;
}

// Primer izdelkov
const izdelki = [
    {ime: "Kruh", cena: 1.2, kolicina: 2},
    {ime: "Mleko", cena: 0.8, kolicina: 1},
    {ime: "Jabolka", cena: 0.5, kolicina: 5}
];

console.log(izracunajSkupnoCeno(izdelki)); // Izpis skupne cene
```

Primer izločitve razreda

```
// Razred, ki upravlja z detajli knjige in njenim inventarjem
class Knjiga {
    constructor(naslov, avtor, zaloga) {
        this.naslov = naslov;
        this.avtor = avtor;
        this.zaloga = zaloga; // število razpoložljivih kopij
    }

    // Prikaz podatkov o knjigi
    prikaziPodatke() {
        console.log(`Naslov: ${this.naslov}, Avtor: ${this.avtor}, Zaloge:
                    ${this.zaloga}`);
    }

    // Sprememba števila zalog
    dodajZaloge(kolicina) {
        this.zaloga += kolicina;
    }
}

// Ustvarjanje in upravljanje knjige
const knjiga = new Knjiga("Vojna in mir", "Lev Tolstoj", 5);
knjiga.prikaziPodatke();
knjiga.dodajZaloge(3); // Dodajanje knjig v zalogo
knjiga.prikaziPodatke();
```

```
class Knjiga { // Razred za upravljanje z detajli knjige
    constructor(naslov, avtor) {
        this.naslov = naslov;
        this.avtor = avtor;
    }

    prikaziPodatke() {
        console.log(`Naslov: ${this.naslov}, Avtor: ${this.avtor}`);
    }
}

class Inventar { // Razred za upravljanje z inventarjem knjige
    constructor() { this.zaloge = {}; }

    // Dodajanje knjig v zalogo
    dodajZaloge(naslov, kolicina) {
        if (!this.zaloge[naslov]) { this.zaloge[naslov] = 0; }
        this.zaloge[naslov] += kolicina;
    }

    // Prikaz zaloge za določeno knjigo
    prikaziZaloge(naslov) {
        console.log(`Zaloge za '${naslov}': ${this.zaloge[naslov] || 0}`);
    }
}

// Ustvarjanje knjige in upravljanje njenega inventarja
const knjiga = new Knjiga("Vojna in mir", "Lev Tolstoj");
const inventar = new Inventar();
inventar.dodajZaloge("Vojna in mir", 5); // Začetna zaloga
knjiga.prikaziPodatke();
inventar.prikaziZaloge("Vojna in mir");
inventar.dodajZaloge("Vojna in mir", 3); // Dodajanje knjig v zalogo
inventar.prikaziZaloge("Vojna in mir");
```

Primer generalizacije

```
class Pes {
    constructor(ime) {
        this.ime = ime;
    }

    zvok() {
        console.log(`${this.ime} pravi: Hov!`);
    }

    jesti() {
        console.log(`${this.ime} je jedel.`);
    }
}

class Macka {
    constructor(ime) {
        this.ime = ime;
    }

    zvok() {
        console.log(`${this.ime} pravi: Mjav!`);
    }

    jesti() {
        console.log(`${this.ime} je jedel.`);
    }
}
```

```
class Zival {
    constructor(ime) {
        this.ime = ime;
    }

    zvok() {
        // Implementacija ni določena tukaj, saj se razlikuje med podrazredi
    }

    jesti() {
        console.log(`${this.ime} je jedel.`);
    }
}

class Pes extends Zival {
    zvok() {
        console.log(`${this.ime} pravi: Hov!`);
    }
}

class Macka extends Zival {
    zvok() {
        console.log(`${this.ime} pravi: Mjav!`);
    }
}

// Uporaba
const rex = new Pes("Rex");
rex.zvok(); // Rex pravi: Hov!
rex.jesti(); // Rex je jedel.

const tom = new Macka("Tom");
tom.zvok(); // Tom pravi: Mjav!
tom.jesti(); // Tom je jedel.
```

Primer enkapsulacije

```
class BancniRacun {
    constructor(imeLastnika, stanje) {
        this.imeLastnika = imeLastnika;
        this.stanje = stanje; // Stanje računa je javno dostopno
    }
}

// Uporaba
const racun = new BankaRacun("Janez Novak", 1000);

// Dvig narejen preko direktne spremembe vrednosti
racun.stanje = racun.stanje - 100;

// Direkten dostop do vrednosti spremenljivke
console.log(`Dvig v višini 100 €. Novo stanje: ${racun.stanje} €.`);
```

```
class BankaRacun {
    #stanje; // Zasebna spremenljivka za stanje računa

    constructor(imeLastnika, zacetnoStanje) {
        this.imeLastnika = imeLastnika;
        this.#stanje = zacetnoStanje; // Dostop do stanja samo preko metod razreda
    }

    // Metoda za pridobivanje trenutnega stanja
    getStanje() {
        console.log(`Trenutno stanje: ${this.#stanje} €.`);
        return this.#stanje;
    }

    // Metoda za polog sredstev
    polog(znesek) {
        this.#stanje += znesek;
        console.log(`Polog v višini ${znesek} €. Novo stanje: ${this.#stanje} €.`);
    }

    // Metoda za dvig sredstev
    dvig(znesek) {
        if (znesek <= this.#stanje) {
            this.#stanje -= znesek;
            console.log(`Dvig v višini ${znesek} €. Novo stanje: ${this.#stanje} €.`);
        } else {
            console.log('Ni dovolj sredstev za izvedbo dviga.');
        }
    }
}

// Uporaba
const racun = new BankaRacun("Janez Novak", 1000);
racun.getStanje(); // Izpis trenutnega stanja
racun.polog(200); // Dodajanje sredstev
racun.dvig(500); // Dvig sredstev
racun.getStanje(); // Izpis trenutnega stanja
```

Profiliranje

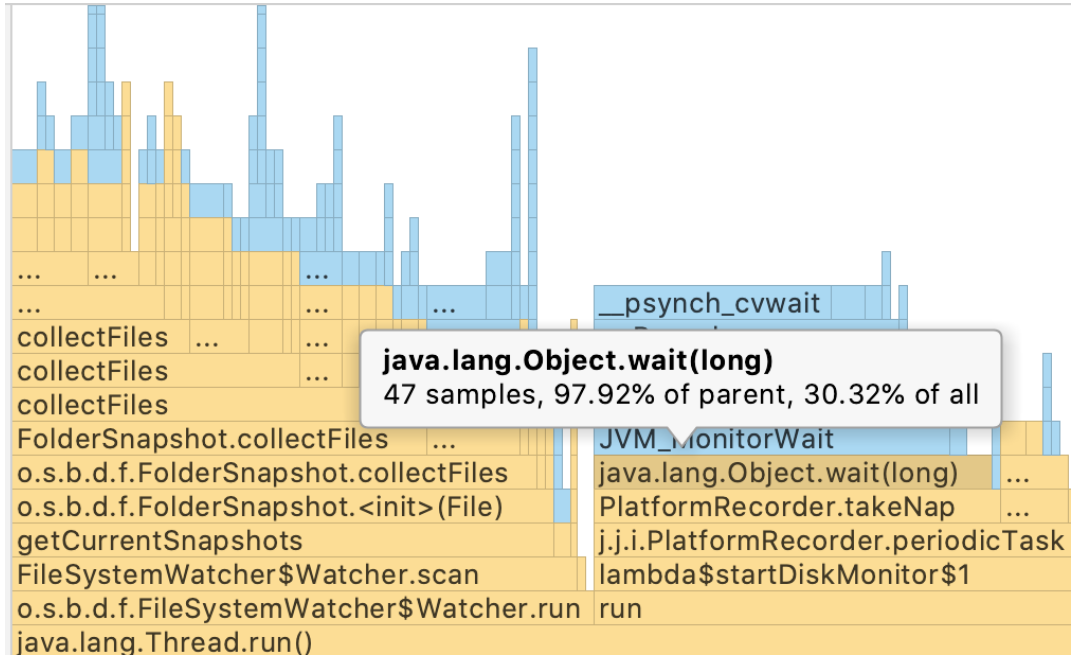
- **Profiliranje** (angl. *profiling*) je postopek spremljanja delovanja programske opreme tekom njenega zagona za namen optimizacije kode. Pri tem se uporabi program **profiler**.

- Pri tem se ustvari poročilo:
  - **Analiza pomnilnika/objektov** – koliko objektov določenega tipa je bilo ustvarjenih in uporabljenih, koliko jih je aktivnih, koliko pomnilnika zasedajo itd. Uporabno za optimizacijo porabe pomnilnika in za iskanje napak pri tvorjenju objektov.

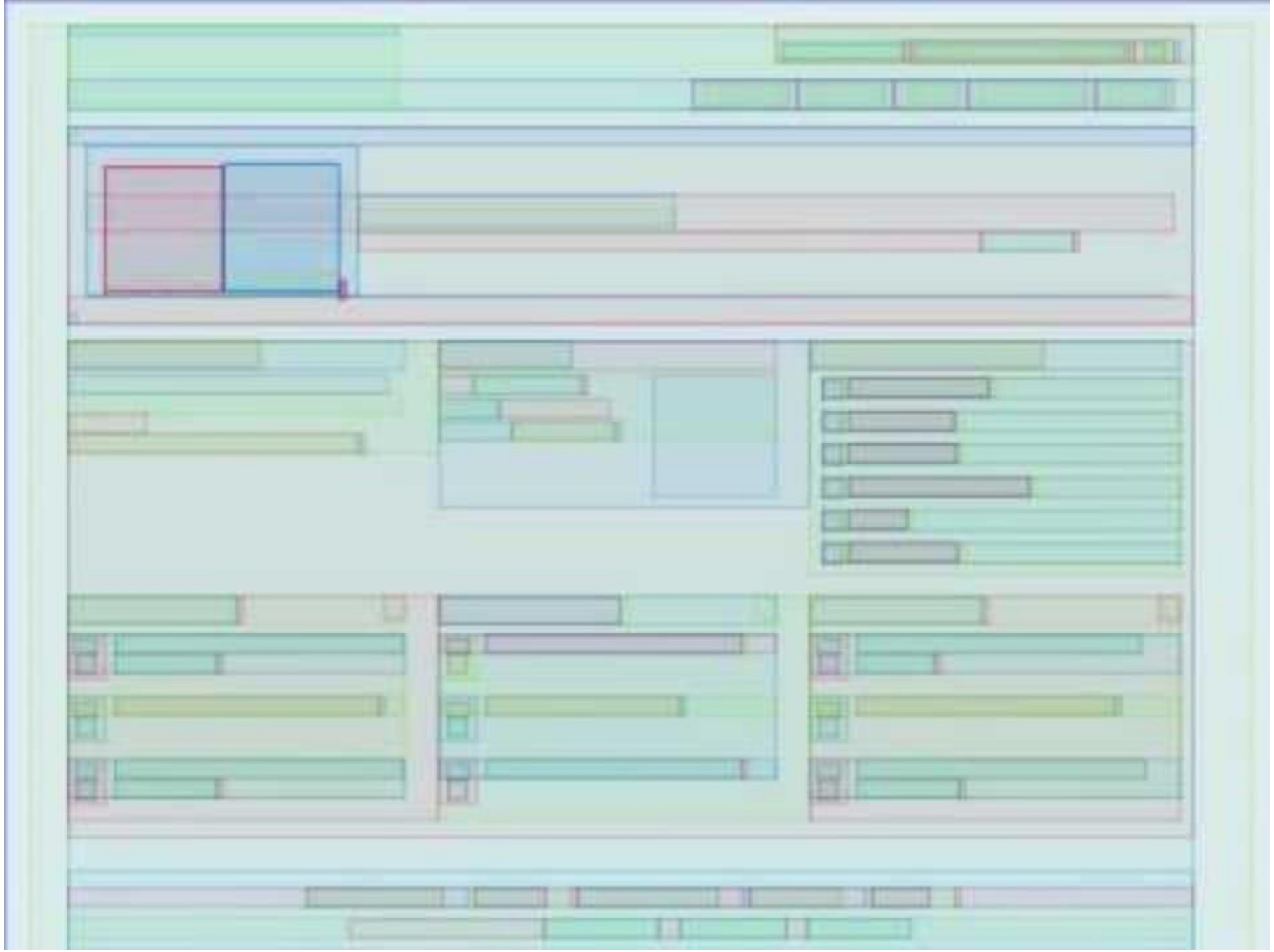
Profiliranje

- **Analiza časa izvajanja** – koliko časa porabijo posamezne metode za izvajanje; prikazuje tudi zgodovino izvajanja metod v neki aplikaciji. Uporabno za iskanje ozkih grl, kje se porabi največ časa pri izvajanju programa – optimizacija hitrosti izvajanja.

- **Pokritost kode** – kaže, kateri deli kode se največkrat kličejo in procentualno koliko kode v aplikaciji se je dejansko izvedlo in s kakšno frekvenco. Daje vpogled v najbolj izvajano področje programa.

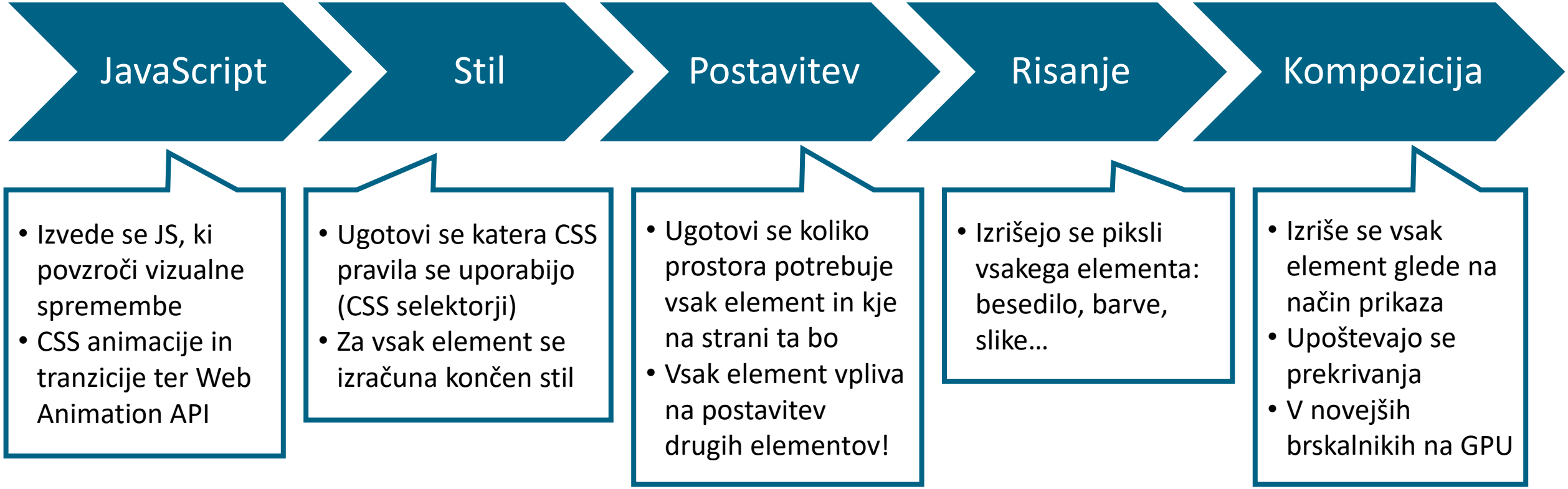


Method	Samples
43.7% org.springframework.boot.devtools.restart.**RestartLauncher.run**()	3,070
30.1% thread_start	2,119
13.1% java.lang.**Thread.run**()	920
3.7% org.springframework.beans.factory.support.**DefaultSingletonBeanRegist**	263
2.3% org.springframework.samples.petclinic.**PetClinicApplication.main**(String[]	163
2.1% jdk.internal.agent.**Agent.startAgent**()	149
1.2% unknown_Java	81
1.0% sun.instrument.**InstrumentationImpl.loadClassAndCallPremain**(String,	72
< 1% Runtime1::monitorenter	57
< 1% com.intellij.rt.execution.application.**AppMainV2$1.run**()	22
< 1% not_walkable_Java	16
< 1% java.lang.invoke.**DirectMethodHandle$Holder.invokeStatic**(Object, Objec	16
< 1% jdk.internal.vm.**VMSupport.serializeAgentPropertiesToByteArray**() → j	15
< 1% no_Java_frame	12
< 1% com.sun.management.internal.**GarbageCollectorExtImpl.createGCNotific**	12



Profiliranje spletnih aplikacij
Pixel layout

- Kaj se zgodi ob tem, ko brskalnik prikaže spletno stran (video):



Akcije brskalnika

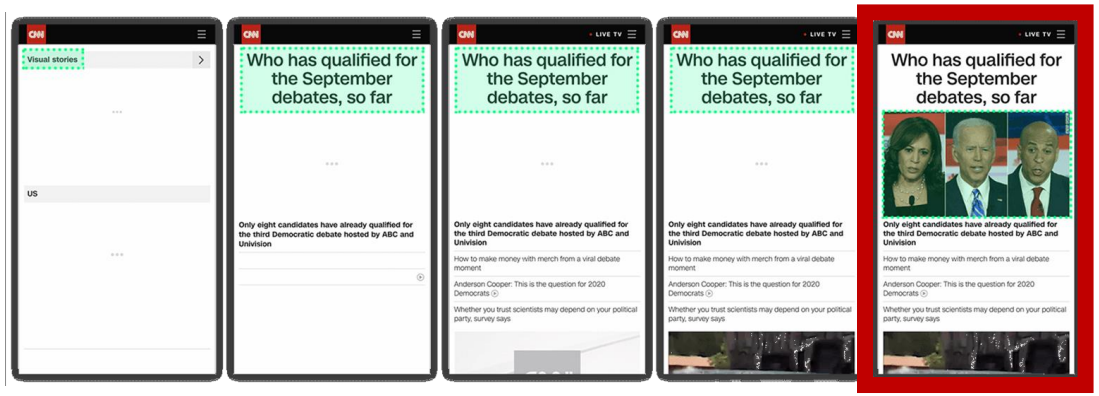
- **Reflow** je akcija, ki se izvede, ko spremembe stanja strani zahtevajo ponovno postavitev elementov.
  - Primeri: sprememba velikosti okna brskalnika, sprememba DOM-a.
  - Izvede se celoten *Pixel layout*:

- **Repaint** je akcija, ki se izvede, ko spremembe stanja strain zahtevajo le ponoven izris (in ne postavitev) elementov.
  - Primeri: sprememba barv in vidnosti elementov.
  - Izpusti se *Postevitev*.

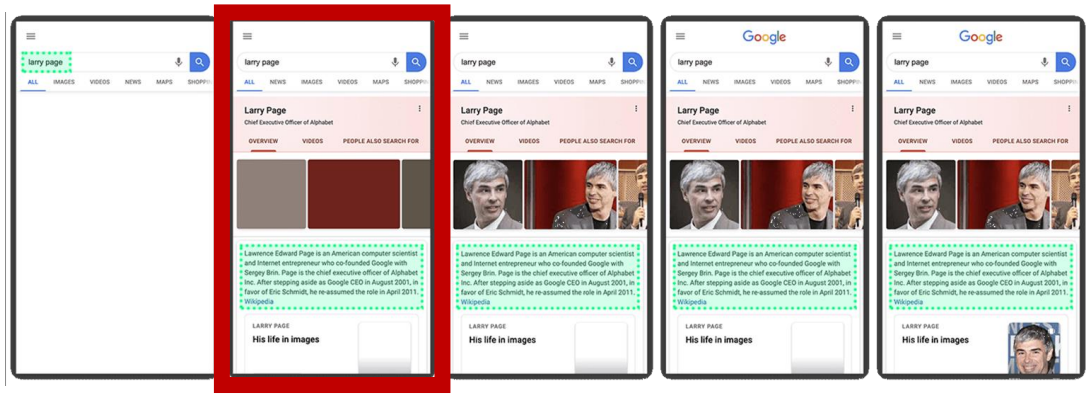
- Zaželjeno je, da naše spremembe strani ne sprožijo *reflow* ali *repaint* akcij.

Opzimitacija izvajanja spletnih aplikacij
Core Web Vital metrics – Hitrost nalaganja

- **LCP** (Largest Contentful Paint) – čas do prikaza *največjega* elementa vsebine (slika, besedilo, kar koli je glavna vsebina spletne aplikacije).
- Primera:



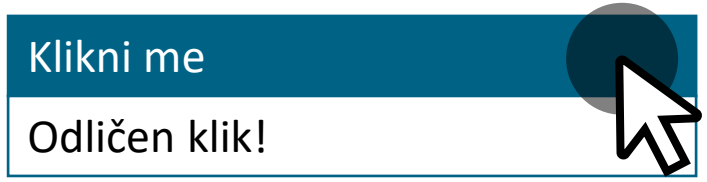
Pozen LCP



Hiter LCP

Opzimitacija izvajanja spletnih aplikacij
Core Web Vital metrics – Odzivnost

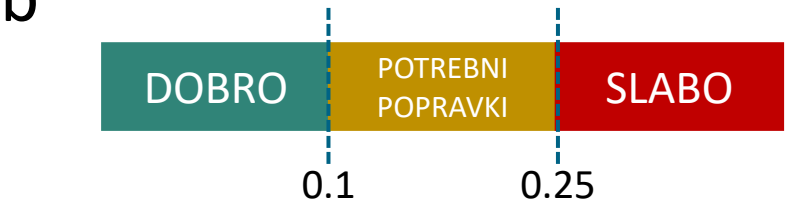
- **FID** (First Input Delay) – čas od *prve* interakcije uporabnika s spletno aplikacijo do odgovora, ki ga vidimo kot spremembo na strani.
  - Zastarel in bo nadomeščen z *INP*.
- **INP** (Interaction to Next Paint) – nadaljši čas od interakcije uporabnika s spletno aplikacijo do odgovora tekom celotne uporabe aplikacije.
- Primer slabe odzivnosti:



2.1 sek

Opzimitacija izvajanja spletnih aplikacij
Core Web Vital metrics – Vizualna stabilnost

- **CLS** (Cumulative Layout Shift) – količina sprememb postavitve tekom nalganja in uporabe spletnih aplikacij.
- Primer nestabilnosti:



Izračuna se kot:
delež premaknjenih elementov *
oddaljenost od prejšnje pozicije