

Testiranje Java aplikacije

- Testiranje Java aplikacije, ne razlikuje se mnogo od testiranja u ostalim tehnologijama, i može se podeliti na:
 - Testiranje performansi
 - Load
 - Stress
 - Unit testiranje
 - Testiranje interfejsa
 - Testiranje ponašanja
 - Testiranje stanja
 - ...
- U nastavku će biti obrađeno unit testiranje u Javi

Šta je unit testiranje

- Unit testiranje je testiranje klasa i metoda (pre svega metoda)
- U unit testu, proveravamo da li metode imaju očekivano ponašanje tako što ih startujemo, a zatim proverimo da li su na osnovu unetih parametara, vratili očekivane vrednosti. Na primer:

```
static int hello(int a, int b){  
    return a+b;  
}  
  
public static void main(String[] args) {  
    int expected = 5, p1 = 2, p2 = 3;  
    int res = hello(p1,p2);  
    if(res==expected){  
        System.out.println("Test passed");  
    } else {  
        System.out.println("Test failed");  
    }  
}
```

Zašto vršimo unit testiranje

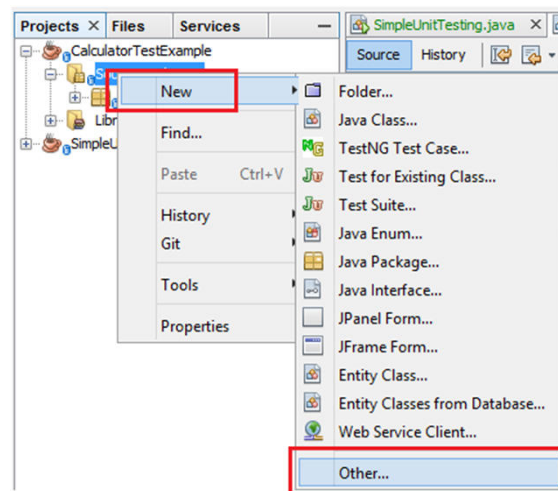
- Unit testiranje je važno najviše zbog toga da ne bismo novim funkcionalnostima projekta poremetili postojeće funkcionalnosti.
- Unit testiranje treba sprovoditi nakon izmena u kodu
- Unit testovi bi trebalo da budu u sastavu projekta, ali dovoljno odvojeni da možemo da ih po potrebi isključimo iz procesa prevođenja
- Za unit testiranje, najčešće se koriste gotova rešenja, radije nego ručno pisana
- Najčešće korišćeno gotovo rešenje za unit testiranje u Javi je JUnit. Ovo okruženje, podržano je i u netbeansu.

JUnit unit testiranje u netbeans-u (jcex122014 CalculatorTestExample)

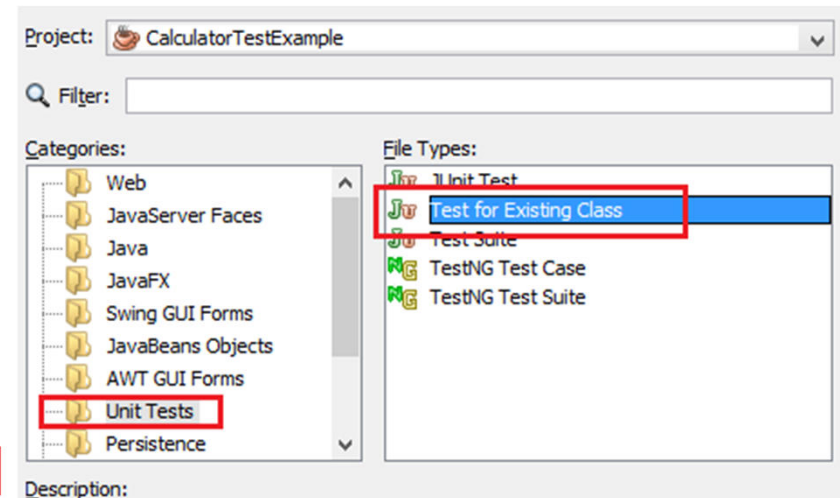
- Ako u projektu imamo sledeću klasu koju želimo da uvrstimo u proces testiranja JUnit frameworkom

```
package calculatortestexample;  
public class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

- Unutar istog projekta treba odabrati opciju new->other



- Zatim Unit Tests->Test for Existing Class



JUnit unit testiranje u netbeans-u

- NetBeans sadrži wizar da za kreiranje JUnit Test klase
- Jedino što je neophodno obezbediti, jeste naziv klase na kojoj će se vršiti testiranje
- Lista opcija (checkbox-ova) na dnu wizar da, predstavlja pomoćne metode koje će wizar d eventualno generisati

The screenshot shows the 'Existing Class To Test' wizard in NetBeans. The 'Class to Test' field is highlighted with a red rectangle and contains the text 'calculatortestexample.Calculator'. To its right is a 'Browse...' button. Below this, the 'Created Test Class' field contains 'calculatortestexample.CalculatorTest'. The 'Project' field contains 'CalculatorTestExample'. The 'Location' dropdown menu is set to 'Test Packages'. The 'Created File' field shows the full path: 'x122014\CalculatorTestExample\test\calculatortestexample\C'. At the bottom, there are three columns of checkboxes: 'Method Access Levels' (Public, Protected, Package Private), 'Generated Code' (Test Initializer, Test Finalizer, Test Class Initializer, Test Class Finalizer), and 'Generated Comments' (Javadoc Comments, Source Code Headers).

Method Access Levels	Generated Code	Generated Comments
<input checked="" type="checkbox"/> Public	<input checked="" type="checkbox"/> Test Initializer	<input checked="" type="checkbox"/> Javadoc Comments
<input checked="" type="checkbox"/> Protected	<input checked="" type="checkbox"/> Test Finalizer	<input checked="" type="checkbox"/> Source Code Headers
<input checked="" type="checkbox"/> Package Private	<input checked="" type="checkbox"/> Test Class Initializer	
	<input checked="" type="checkbox"/> Test Class Finalizer	

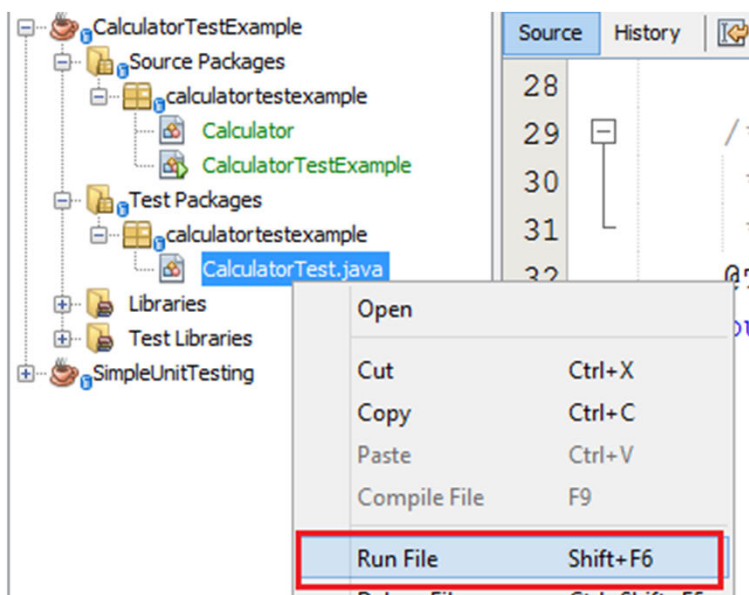
JUnit unit testiranje u netbeans-u

- Nakon potvrde u wizardu, NetBeans kreira nekoliko metoda (pomoćnih – za eventualnu intervenciju na sistemu pre, posle i u toku izvršenja testa) i test metod(e) koji odgovaraju metodama testirane klase
- Takođe, NetBeans automatski generiše promenljive za ulazne i izlazne parametre, kojima treba dodeliti očekivane vrednosti
- Anotacija @Test označava da će metod koji sledi biti tretiran kao test metod

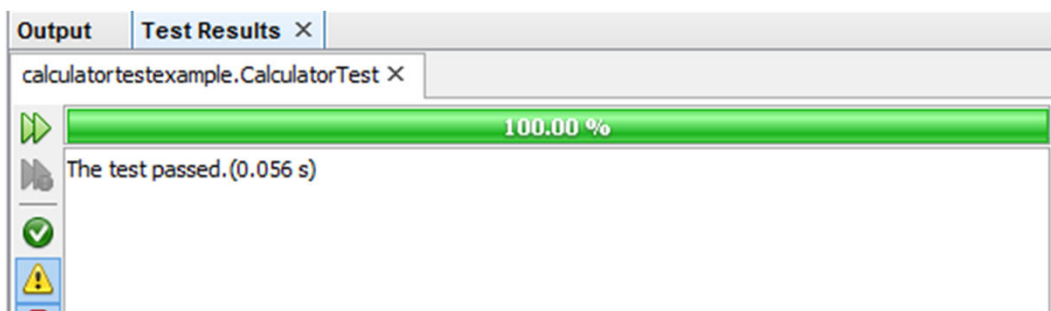
```
@Test
public void testAdd() {
    System.out.println("add");
    int a = 2;
    int b = 3;
    Calculator instance = new Calculator();
    int expectedResult = 5;
    int result = instance.add(a, b);
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to fail.
    //fail("The test case is a prototype.");
}
```

JUnit unit testiranje u netbeans-u

- Kada je metod za testiranje pripremljen treba odabrati **Run File** opciju kontekstnog menija test klase kako bi test bio startovan

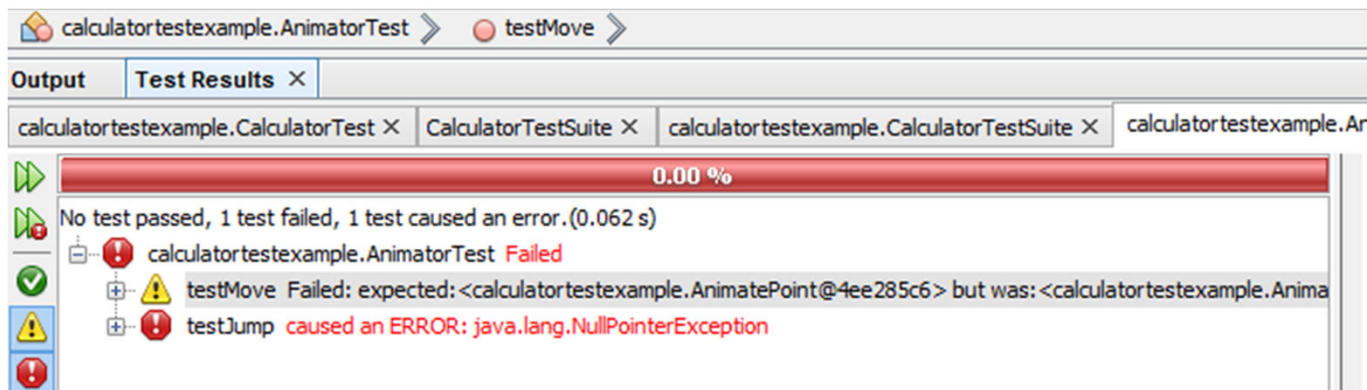


- Nakon startovanja, u donjem delu NetBeans-a, pojavljuje se novi panel (Test Results) sa rezultatima testiranja



JUnit unit testiranje u netbeans-u

- Ukoliko test iz nekog razloga ne uspe, NetBeans će u test panelu objaviti da je došlo do greške i probati da je identifikuje



Testiranje privatnih metoda

- Za testiranje privatnih članova klase koristi se refleksija (sama refleksija će biti detaljno obrađena u kasnijim lekcijama)

```
@Test
public void testMul() throws NoSuchMethodException, IllegalAccessException, IllegalArgumentException, Invo←
cationTargetException {
    System.out.println("mul");
    Integer a = 2;
    Integer b = 3;
    int expectedResult = 6;
    Calculator instance = new Calculator();
    Method method = Calculator.class.getDeclaredMethod("mul", new Class[]{int.class, int.class});
    method.setAccessible(true);
    int result = (int)method.invoke(instance, 2, 3);
    assertEquals(expectedResult, result);
}
```

JUnit test suite

- Umesto da svaku kreiranu test klasu startujemo ponaosob, možemo kreirati klasu koja će biti zadužena za startovanje svih test klasa
- Ovakva klasa, naziva se test suite
- Sve što je potrebno za kreiranje test suite klase jeste označavanje klase anotajama `RunWith` i `Suite.SuiteClasses` sa priloženim nizom test klasa

```
package calculatortestexample;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    calculatortestexample.CalculatorTest.class,
    calculatortestexample.AnimatorTest.class
})
public class CalculatorTestSuite {
}
```

JUnit anotacije - @Test

- Svaki metod iznad koga stavimo anotaciju @Test, biće uzet u obzir prilikom izvršavanja unit testa
- Test klasa mora imati bar jedan ovakav metod, inače će doći do greške
- Anotacija @Test može biti parametrizovana jednim od dva paramtra: **timeout** i **expected**

```
@Test(expected = ArithmeticException.class)
public void expectedExample() throws InterruptedException {
    System.out.println("It's ok. I expect division by zero.");
    int x = 10/0;
}

@Test(timeout = 1001)
public void timeoutExample() throws InterruptedException {
    System.out.println("I will pass. Timeout is ok");
    Thread.sleep(1000);
}
```

JUnit anotacije

@Before – Metod će biti startovan pre poziva svake test metode

@BeforeClass – Metod (obavezno statički) će biti startovan pre svih testova te klase

@After – Metod će biti startovan nakon poziva svake test metode

@AfterClass – Metod (obavezno statički) će biti startovan nakon svih testova te klase

@Ignore – Metod se ne startuje, iako ima anotaciju @Test

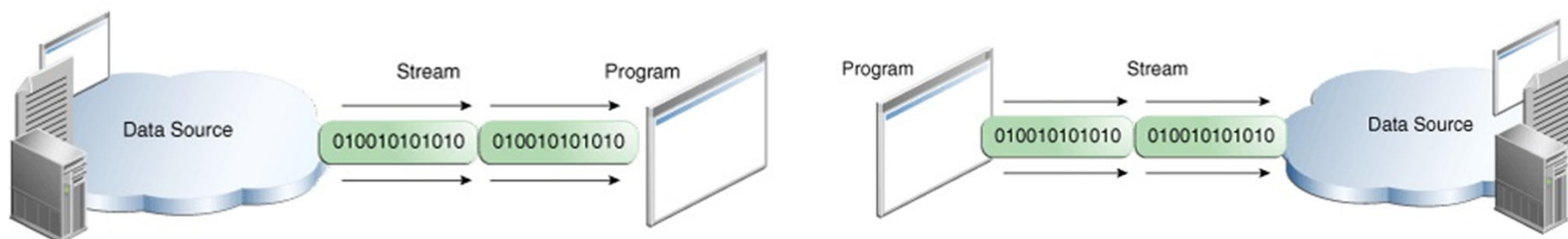
```
@AfterClass
public static void afterClassExample() {
    System.out.println("This will be started after all tests in this class");
}
@Before
public void beforeExample() {
    System.out.println("This will be started before every test");
}
```

JUnit assertion metode

- Assert metode su metode za proveru određenog stanja. Do sada smo koristili metod **assertEquals**, koji proverava da li jedna vrednost odgovara drugoj. Ali takođe, postoje i druge assert metode:
- <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>
- Sve assert metode nalaze se u klasi **org.junit.Assert**

Tokovi podataka

- Podaci ulaze u aplikaciju ili izlaze iz nje, putem **toka** (stream)
- Tokove koristimo već duže vreme, iako smo možda nesvesni toga, jer je čak i emitovanje podataka na izlaz (na primer, naredba `println`), u stvari slanje podataka u neki tok
- Podaci se kroz tok kreću sekvencijalno – bajt po bajt.

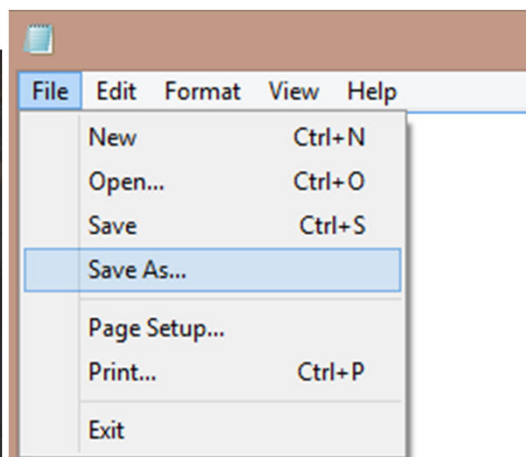
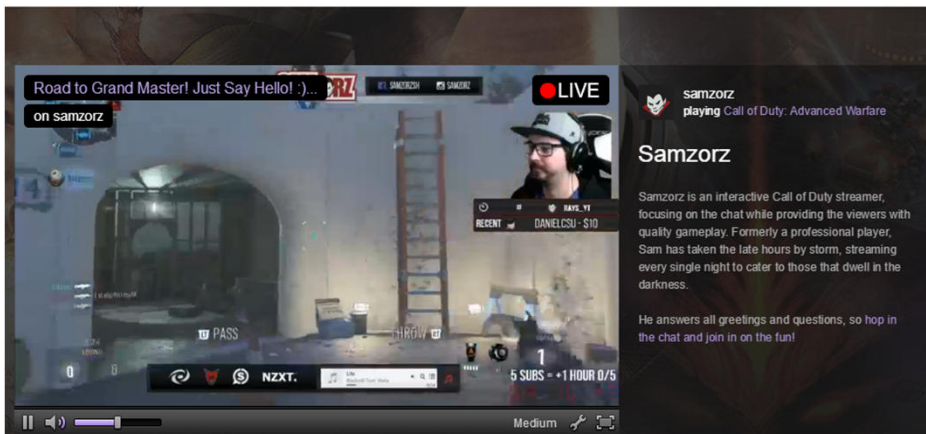


Gde se sve koriste tokovi?

- Prilikom transporta podataka preko mreže
- Prilikom bilo kakvog čitanja ili pisanja u fajlove
- Prilikom komunikacije sa računara sa hardverom
- ...

twitch

Search Browse Go Turbo Log In Sign Up

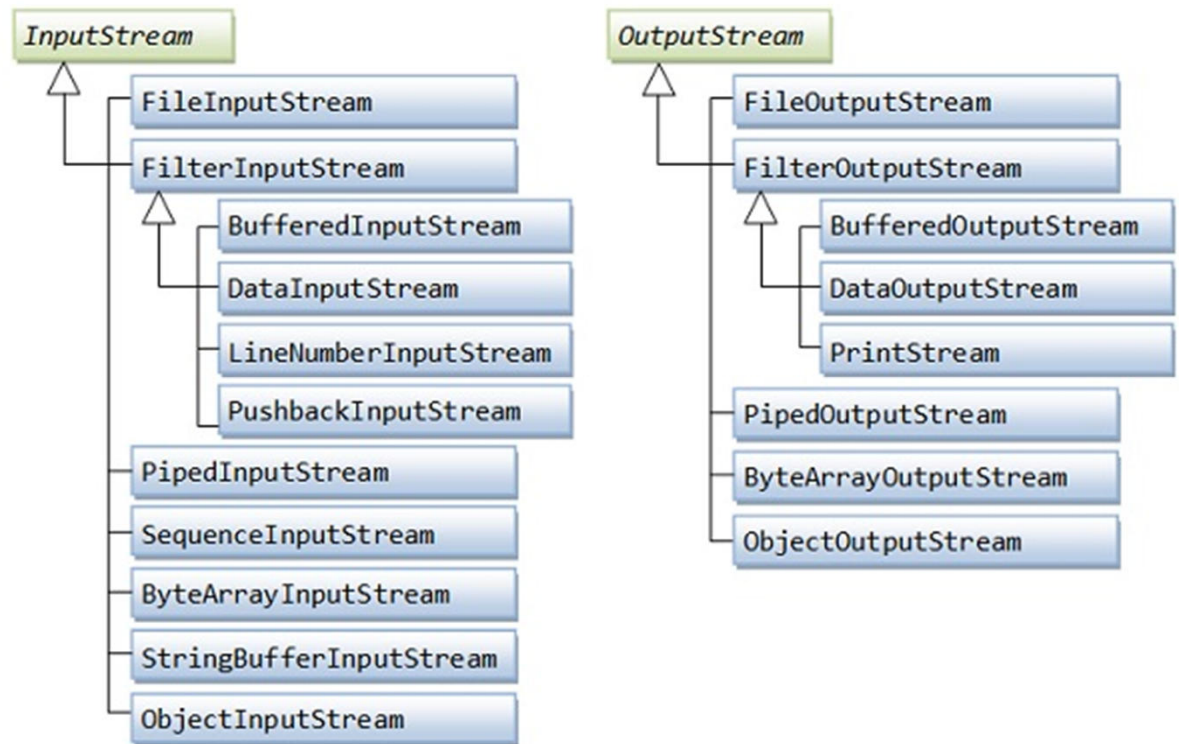


Tokovi podataka

LINKgroup

Ugrađene klase za rad sa IO tokovima

- Sve ugrađene klase za rad sa io tokovima u Javi, nalaze se u paketu **java.io**
- Dve bazne klase za rad sa tokovima u Javi su **InputStream** i **OutputStream**




InputStream i OutputStream

- Klase InputStream i OutputStream su apstraktne klase koje se koriste kao osnova u kreiranju klasa za rukovanje tokovima
- Naslednici ove dve klase moraju implementirati (**respektivno**) metod za čitanje (**read**) strima i metod za pisanje (**write**) u strim.
- U primeru, kreirana je klasa za koja nasleđuje klasu OutputStream
- Klasa ima implementaciju metode write, koja prihvata bajt i prikazuje ga na izlazu
- Ulazni parametar nije pravi bajt već četvorobajtna sekvenca bitova
- Pitanje je šta ta sekvenca predstavlja za nas?

```
class MyStream extends OutputStream{
    @Override
    public void write(int b) throws IOException {
        System.out.println("Writing to stream : " + Integer.toBinaryString(b));
    }
}

MyStream ms = new MyStream();
ms.write(1);
ms.write(2);
ms.write(3);
```



InputStream i OutputStream

- Ako bi svaku jedinicu prethodno kreiranog strima tretirali kao jedan broj, mogli bi reći da radimo sa strimom celobrojnih vrednosti. Tada bi sadržaj strima bio 1, 2 i 3 (upravo ono što smo i uneli)
- Takođe, mogli bi reći da će svaka jedinica toka (jedan ceo broj), predstavljati seriju od četiri karaktera, i tada bi sadržaj našeg strima bio serija praznih karaktera. Ali bi zato sledeći unos:

```
ms.write(1970496882);
```
- Proizveo reč niz karaktera koji čini reč **user**
- (jcex132014 SimpleStreamWriter)

Byte stream-ovi

- Bajt strimovi su strimovi u kojima je svaka jedinica strima jedan bajt
- Bajtove preuzimamo iz strima i interpretiramo na način koji nam najviše odgovara, odnosno koji odgovara našem sistemu
- Kada hoćemo da radimo sa podacima na najnižem nivou, tada koristimo ovu vrstu. Na primer, kada hoćemo da kopiramo fajlove koji čiji sadržaj nije tekstualan

FileInputStream i FileOutputStream (jcex132014 ByteStreams)

- Ova dva strima su IO byte strimovi
- Čitaju i pišu u fajlove bajt po bajt
- Sledeći primer, upisuje bajt po bajt u fajl

```
FileOutputStream fos = new FileOutputStream("testfile.txt");  
fos.write(104);  
fos.write(101);  
fos.write(108);  
fos.write(108);  
fos.write(111);  
fos.close();
```

- “Sirovi” sadržaj ovog fajla predstavljaju bajtovi (brojevi od 0-255), ali ako ga otvorimo, videćemo reč **hello**

FileInputStream i FileOutputStream

- Čitanje fajlova pomoću klase FileInputStream je nešto teže nego pisanje. Bar ukoliko hoćemo da čitanje bude automatizovano:

```
FileInputStream fis = new FileInputStream("testfile.txt");
System.out.println(fis.read());
System.out.println(fis.read());
System.out.println(fis.read());
System.out.println(fis.read());
fis.close();
```

Čitanje ručno
Bajt po bajt

```
FileInputStream fis1 = new FileInputStream("testfile.txt");
int b;
while ((b=fis1.read()) != -1) {
    System.out.println(b);
}
fis1.close();
```

Čitanje
automatski

Flush i close

- Svaki tok koji koristi neke eksterne resurse (fajlove ili fajl deskriptore), treba da bude zatvoren nakon upotrebe
- Ukoliko se strim ne zatvori, upotrebljeni resurs će biti nedostupan za ostale delove sistema (osim ako sistem ne dozvoljava suprotno)
- Metod **close** zatvara postojeći tok i potvrđuje upis (tok nije više upotrebljiv, dok ponovnog otvaranja)

```
fos.close();
```
- Ako hoćemo samo da potvrdimo upis, ali da ne zatvorimo tok, koristimo metod **flush**

```
fos.flush();
```

Karakter strimovi

- Tehnički, svi strimovi su bajt strimovi, jer se svi digitalni podaci predstavljaju bajtovima, jedini je problem u njihovom rasporedu i onome šta predstavljaju.
- Na prethodnom slajdu, rukovali smo tekстом pomoću byte stream-a. Umesto toga, mogli smo koristiti character stream klase
- Character stream-ovi su nam naročito interesantni, jer ćemo tokom pravljenja programa, dosta raditi sa tekстом
- Na najnižem nivou karakter stream-ova su klase **Writer** i **Reader**
- Iznad ovih klasa su klase **InputStreamReader** i **OutputStreamWriter**. One kombinuju mogućnost ubacivanja teksta u tok, ali moraju imati već obezbeđen tok
- Iznad njih mogu biti razne klase. Na primer: **FileReader** i **FileWriter**

Čitanje i pisanje fajlova pomoću klasa `FileReader` i `FileWriter`

- Klase `FileWriter` i `FileReader` imaju mogućnost upisa u fajl ili čitanja iz njega. Prilikom upisa, mogu se koristiti bajtovi ili direktno `String` tip, dok se prilikom čitanja koriste isključivo bajtovi
- Bajtovi koji se dobijaju prilikom čitanja, predstavljaju karaktere u unicode reprezentaciji (2 bajta po karakteru).

```
FileWriter fw = new FileWriter("testfile.txt");
fw.write("Hello world");
fw.close();
FileReader fr = new FileReader("testfile.txt");
int b;
while ((b=fr.read()) != -1)
    System.out.print((char)b);
fr.close();
```

(jcex132014 `CharacterStream`)

Baferovano i nebaferovano rukovanje strimovima

- **Nebaferovano** upravljanje podacima podrazumeva sekvencijalno čitanje/upis i obradu na nivou najmanje jedinice
- **Baferovano** upravljanje podacima podrazumeva smeštanje određene količine podataka (najčešće onoliko koliko predstavlja neku logičku celinu) u memoriju pre obrade ili kompletnu obradu pre slanja u tok
- Baferovano upravljanje podacima često se koristi prilikom rada sa grafikom i fajlovima
- Oba načina imaju svoje dobre i loše strane
 - Baferovani sistem je loš ako ne znamo šta tražimo u toku, ali je dobar ako nam treba kompletan sadržaj toka
 - Nebaferovani sistem je loš ako se ono što tražimo nalazi na kraju toka, ali je dobar ako nismo sigurni gde se nalazi a ne treba nam kompletan sadržaj toka
- Način na koji smo do sada tretirali strimove je bio **nebaferovan**

Beforovani i nebaferovani tokovi u Javi (jcex132014 BufferedStreams)

- Baferovani tokovi nisu pravi tokovi, već više okviri za tokove
- U Javi se baferovani strimovi realizuju kroz četiri klase: **BufferedInputStream**, **BufferedOutputStream** za bajt strimove i **BufferedReader** i **BufferedWriter** za strimove karaktera
- Kod baforovanog upisa flush ima veoma važnu ulogu, jer će, ukoliko ne izvršimo flush eksplicitno, bafer biti automatski flush-ovan onda kada to bude „odgovaralo“ objektu kojim je predstavljen
- Baferovani tokovi su dobri za čitanje kompletnih linija ili nekih drugih strimovanih grupa podataka

Data streams (jcex132014 DataStreams)

- Data Stream je pojam kojim se predstavlja specijalna vrsta toka, koji je u stanju da prepozna proste tipove podataka prilikom pisanja ili čitanja.
- Svaka Data stream klasa mora implementirati interfejs **DataInput** ili **DataOutput**. Ovi interfejsi su već implementirani u klase **DataOutputStream** i **DataInputStream**
- U data streamovima ne postoji detekcija tipova, pa moramo tačno znati na koji su način reprezentovani podaci unutar toka, kako bi ispravno mogli da ih preuzimamo
- Data streams su dakle dobri, ukoliko hoćemo da radimo sa primitivnim tipovima čiji raspored dobro poznajemo ili kada radimo sa podacima istog tipa

Data streams – Čítanje / upis

- Klada DataOutputStream nije sama po sebi tok, već samo omotač, pa je, da bi mogla da se koristi, neophodno proslediti joj odgovarajući baferovani ili nebaferovani tok.
- Ovaj tok mora biti tok bajtova

```
FileOutputStream fos_diff_type = new FileOutputStream("testfile.txt");  
BufferedOutputStream bos_diff_type = new BufferedOutputStream(fos_diff_type);  
DataOutputStream dos_diff_type = new DataOutputStream(bos_diff_type);  
dos_diff_type.writeInt(10);  
dos_diff_type.writeFloat(15.5f);  
dos_diff_type.writeChar(65);  
dos_diff_type.close();
```

Data streams – Automatsko čitanje

- Metode za čitanje podataka u Data stream-u, ne vraćaju -1 već kada dođe do kraja fajla, izbacuju IOException

```
FileInputStream fis_auto_read = new FileInputStream("testfile.txt");
BufferedInputStream bis_auto_read = new BufferedInputStream(fis_auto_read);
DataInputStream dis_auto_read = new DataInputStream(bis_auto_read);
try {
    while(true) {
        System.out.println(dis_auto_read.readInt());
    }
} catch (EOFException e) {
    System.out.println("Stream reading finished");
} finally {
    dis_auto_read.close();
}
```

Strimovi objekata

- Strimovi objekata su naročito atraktivni, jer omogućavaju strimovanje kompletnih objekata
- Ovaj proces se još naziva i **serijalizacija** (dok je čitanje objekata iz strima **deserijalizacija**)
- Da bi objekat mogao da bude strimovan mora da implementira interfejs **Serializable**.
- Većina ugrađenih Java klasa ispunjava ovaj uslov, ali prilikom kreiranja sopstvene klase, ukoliko želimo da omogućimo njenu serijalizaciju, moramo implementirati ovaj interfejs
- Implementacija Serializable interfejsa, ne zahteva implementaciju metoda

Vežba 1 (jcex132014 UsersExample)

- Dat je fajl users.txt, sa sledećim sadržajem:

```
id:01|firstName:Petar|lastName:Petrovic|jmbg:1234567890123  
id:02|firstName:Jovan|lastName:Jovanovic|jmbg:1234567890124  
id:03|firstName:Nikola|lastName:Nikolic|jmbg:1234567890125
```

- Potrebno je napraviti aplikaciju koja će pročitati fajl i emitovati njegov sadržaj u sledećem formatu:

```
User Id: 01  
First name: Petar  
Last name: Petrovic  
Jmbg: 1234567890123
```

```
-----  
User Id: 02  
First name: Jovan  
Last name: Jovanovic  
Jmbg: 123456789024
```

Vežba 2 (jcex132014 MergeFiles)

- Potrebno je napraviti aplikaciju koja konkatiniira fajlove (sastavlja dva fajla u jedan). Nazivi fajlova su prviFajl.txt i drugiFajl.txt. Potrebno je sastaviti sadržaje ova dva fajla i smestiti ih u fajl sa nazivom treciFajl.txt.