

Java'da Hata Ayıklama

Contents

Giriş	3
Derleme zamanı hatalarını bulma	4
Kodu derle	4
Diger örnekler	7
IDE'nin Dil Sunucusu Protokolü	13
Çalışma zamanı ve mantıksal hataları bulma	19
Yazdırma ifadeleri	19
Birim test veya test durumları	23
Hata Ayıklayıcılar	25
Hata ayıklayıcılarının temelleri	25
Gelişmiş Hata Ayıklama	38

Sonuç	39
Belgelendirme	42

Giriş

İlk denemede her zaman mükemmel kod yazmıyoruz, değil mi?

Me: *Writes some code*

Software:



Figure 1: mükemmel kod

Bu öğreticide, kodumuzdaki hataları anlamadan ve nasıl düzeltceğimizin yolunu öğreneceğiz.

Derleme zamanı hatalarını bulma

Kodu derle

Java derleyicisi, kodunuzdaki sorunları size bildirmenin harika bir yoluna sahiptir. Size satır numarasını ve sorunun türünü söyleyecek. Ayrıca aynı satırda birden fazla sorun olup olmadığını da belirtecek.

Örnek:

```
class punchcard {  
    public static void main(String[] args) {  
        System.out.println("Hello World!")  
    }  
}
```

Cıktı:

```
/home/suren/Templates/Java/test/test.java:3: error: ';' expected  
    System.out.println("Hello World!")  
               ^
```

```
1 error
```

```
error: compilation failed
```

```
shell returned 1
```

- Çıktıyı inceleyelim ve ne anlama geldiğini anlayalım.

```
/home/suren/Templates/Java/test/test.java
```

Bu, hataya sahip olan dosyanın yoludur.

Bu, hataya sahip olan dosyayı tanımlamanıza yardımcı olur.

Dosya adını görmek için sonuna bakın.

Dosya adı **test.java**'dır.

- Çıktiya devam edelim.

```
:3: error:
```

Bu çıktıının bu kısmı hatanın satır numarasını söyler.

IDE'nin editörünün sol tarafındaki satır numarasını görebilirsiniz.

Satır numarasını **3** olarak görebilirsiniz.

- Çıktiya devam edelim.

```
' ; ' expected
```

Bu çıktıının bu kısmı sorunun ne olduğunu söyler.

Bu durumda, size bir **noktalı virgül** eksik olduğunu söylüyor.

Alabileceğiniz birçok başka hata var. Bu hataların bazıları kolay anlaşılırken bazıları kriptiktir. :b

Hatanın anlaşılmadığı bir durumda, **internet** üzerinde arayabilirsiniz. Unutmayın, bir programcı olarak Internet en iyi dostunuzdur.

- Çıktiya devam edelim.

```
System.out.println("Hello World!")
```

Bu, Java derleyicisinin muhteşem bir özelliğidir. Hatanın tam konumunu size söyler.

Bu sayede uzun bir kod satırında hatanın nerede olduğunu merak etmenize gerek kalmaz.

- Çıktiya devam edelim.

```
1 error
```

Bu çıktıının bu kısmı kodunuzdaki hataların sayısını söyler.

Evet, Java derleyicisi kodunuzda birden fazla hata olup olmadığını size söyleyebilir.

Bu durumda, yalnızca bir hata vardır.

- Çıktiya devam edelim.

```
error: compilation failed
```

Bu çıktıının bu kısmı hatanın türünü söyler.

Bu durumda, derlemenin başarısız olduğunu söylüyor.

- Çıktıya devam edelim.

```
shell returned 1
```

Bu satır Java derleyicisinin çıktısının bir parçası değildir.

Bu satır, kabuğun çıktısıdır.

Ancak kabuğun hala 1 döndüğünü bilmek yine de faydalıdır.

0 değeri dönüş, programın başarıyla çalıştığını gösterir.

1 değeri dönüş, programın çalışmada başarısız olduğunu gösterir.

Burada 1 değeri, programın derlemede başarısız olduğunu gösterir.

Diger örnekler

Örnek:

```
class punchcard {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Cıktı:

```
/home/suren/Templates/Java/test/test.java:5: error: reached end of file while parsing
}
^
1 error
error: compilation failed
```

dosya adı: test.java

satır numarası: 5

hata: parsing sırasında dosya sonuna ulaşıldı

hata sayısı: 1

hata türü: derleme başarısız

eksik: }

Örnek:

```
class punchcard {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Çıktı:

```
/home/suren/Templates/Java/test/test.java:3: error: ')' or ',' expected
    System.out.println("Hello World!");
                                         ^
1 error
error: compilation failed
```

dosya adı: test.java

satır numarası: 3

hata: ')' veya ',' bekleniyor

hata sayısı: 1

hata türü: derleme başarısız

eksik:)

Örnek:

```
class punchcard {
    public static void main(String[] args) {
        System.out.println("Hello World!");
        not_read_code_sitting_here;
    }
}
```

Cıktı:

```
/home/suren/Templates/Java/test/test.java:4: error: not a statement
    not_read_code_sitting_here;
               ^
1 error
error: compilation failed
```

dosya adı: test.java

satır numarası: 4

hata: bir ifade değil

hata sayısı: 1

hata türü: derleme başarısız

bu kod geçerli bir ifade değil.

Örnek:

```
class punchcard {
    public static void main(String[] args) {
        System.out.println("Hello World!")
        System.out.println("Hello Second World!");
    }
}
```

Cıktı:

```
/home/suren/Templates/Java/test/test.java:3: error: ';' expected
    System.out.println("Hello World!")
                                         ^
/home/suren/Templates/Java/test/test.java:4: error: unclosed string literal
    System.out.println("Hello Second World!");
                                         ^
2 errors
error: compilation failed
```

dosya adı: test.java

satır numarası: 3 ve 4

hata: ';' bekleniyor ve kapatılmamış dize alıntısı

hata sayısı: 2

hata türü: derleme başarısız

eksik: ; ve "

He couldn't sleep for 2 days
because he missed her.

I couldn't sleep for 4 days
because I missed a stupid " ; " in
my code.

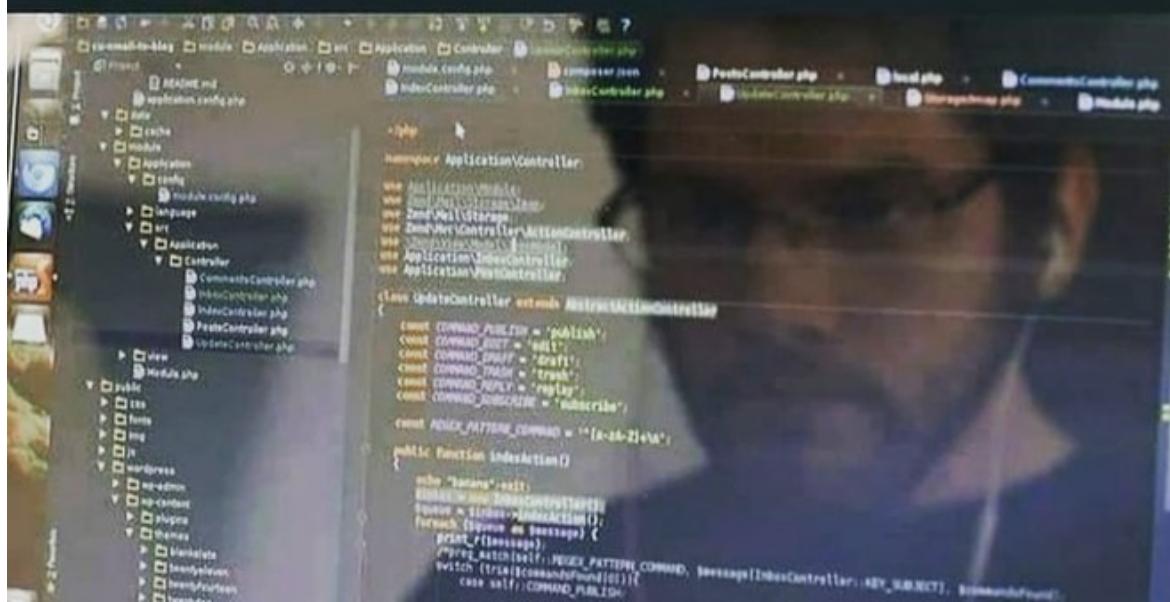


Figure 2: hatalar

IDE'nin Dil Sunucusu Protokolü

En popüler IDE'ler **Dil Sunucusu Protokolü**'nü destekler.

Bu özellik, kodunuzda yazarken hataları görmemizi sağlar.

Bu özellikle kodunuzu derlemeden hataları görebilmenizi sağlar.

Bir LSP'nin, kodunuzdaki hataları göstermek için birçok göstergesi vardır.

- Gezgin Paneli

Sol taraftaki gezgin paneline bakın. Dosya adının yanında bir sayı var.

Bu sayı, dosyadaki hata sayısını size gösterir.

- Mini Harita

Şekil 3'ün sağ üstündeki minik haritaya bakın. Minik haritada kırmızı bir çizgi var.

Bu kırmızı çizgi, dosya içindeki hatanın genel konumunu gösterir.

Bu, uzun bir dosyadaki hataları bulmaya yardımcı olabilir.

- Hata Paneli

IDE'nin alt kısmında bir panel var. Bu panelin birçok kullanışlı sekmesi var. Şimdiye kadar sadece **TERMINAL** sekmesini kullandık.

Şimdi, **PROBLEMLER** sekmesine bakalım.

The screenshot shows a Java code editor interface. At the top, there's a navigation bar with icons for back, forward, search, and more. Below it is a toolbar with tabs for EXPLORER, TEST, and other options. The TEST tab is currently active, showing a list of Java files: .vscode, digit.displayer.java, digit.displldyer.2.java, donut.java, test.java (selected), and test2.java.

The main workspace displays the content of the selected file, `test.java`. The code is:

```
1 class punchcard {  
2     Run | Debug  
3         public static void main(String[] args) {  
4             System.out.println("Hello World!");  
5         }  
6 }
```

A red squiggly underline is under the closing brace of the `main` method, indicating a syntax error. A tooltip above the cursor says "Syntax error, insert ";" to complete BlockStatements Java(1610612976) [Ln 3, Col 36]".

At the bottom, there's a PROBLEMS tab with 6 items, an OUTPUT tab, a DEBUG CONSOLE tab, a TERMINAL tab, and PORTS tab. The PROBLEMS tab is active, showing the error message for `test.java`.

Figure 3: LSP'li temel IDE

Bu sekme, kodunuzdaki hataları liste formatında gösterir.

Kırmızı çarpı-daireye tıklayarak hataya sahip olan satırı gidebilirsiniz.

Listede bulunan hatayı sağ tıklayarak hatayı kopyalayabilirsiniz. Bu, hatayı internette aramak isterseniz faydalı olabilir.

Hata mesajının sonunda bir dizi sayı var.

[Satır 3, Sütun 36]

‘Ln’, hatanın satır numarasını belirtir. (satır 3) ‘Col’, hatanın sütun numarasını belirtir. (sütun 36)

- Kırmızı Alt Çizgi

Şekil 3'teki kırmızı alt çizgiye bakın. Tam olarak 3. satırdaki ')' altında.

Kırmızı alt çizginin üzerine geldiğinizde, hatanın mesajını göreceksiniz. Bazı IDE'ler ayrıca hatayla ilgili daha fazla bilgi gösterebilir. Şekil 4'e bakın.

Bu hataya yönelik bir hızlı düzeltme bile mevcut. Hızlı düzeltme, hatayı düzeltmek için size bir öneri sunar. Kodla uğraşmanıza gerek olmayabilir.

The screenshot shows a Java project in the Explorer view with files like .vscode, digit.displayer.java, digit.displldyer.2.java, donut.java, test.java (selected), and test2.java. The code editor displays a Java file named test.java with the following content:

```
1 class punchcard {  
2     public sta  
3         System.out.println("Hello World!);  
4     }  
5 }  
6
```

A tooltip is shown over the closing parenthesis of the println statement, indicating a "String literal is not properly closed by a double-quote" error from Java (1610612995). Below the editor, the Problems view shows one issue for test.java, which is the same error message. The status bar at the bottom indicates "Showing 1 of 6".

Figure 4: LSP ve hızlı düzeltme olan IDE

Bu araçlarla kodunuzdaki herhangi bir çalışma zamanı hatasını kolayca bulabiliriz.

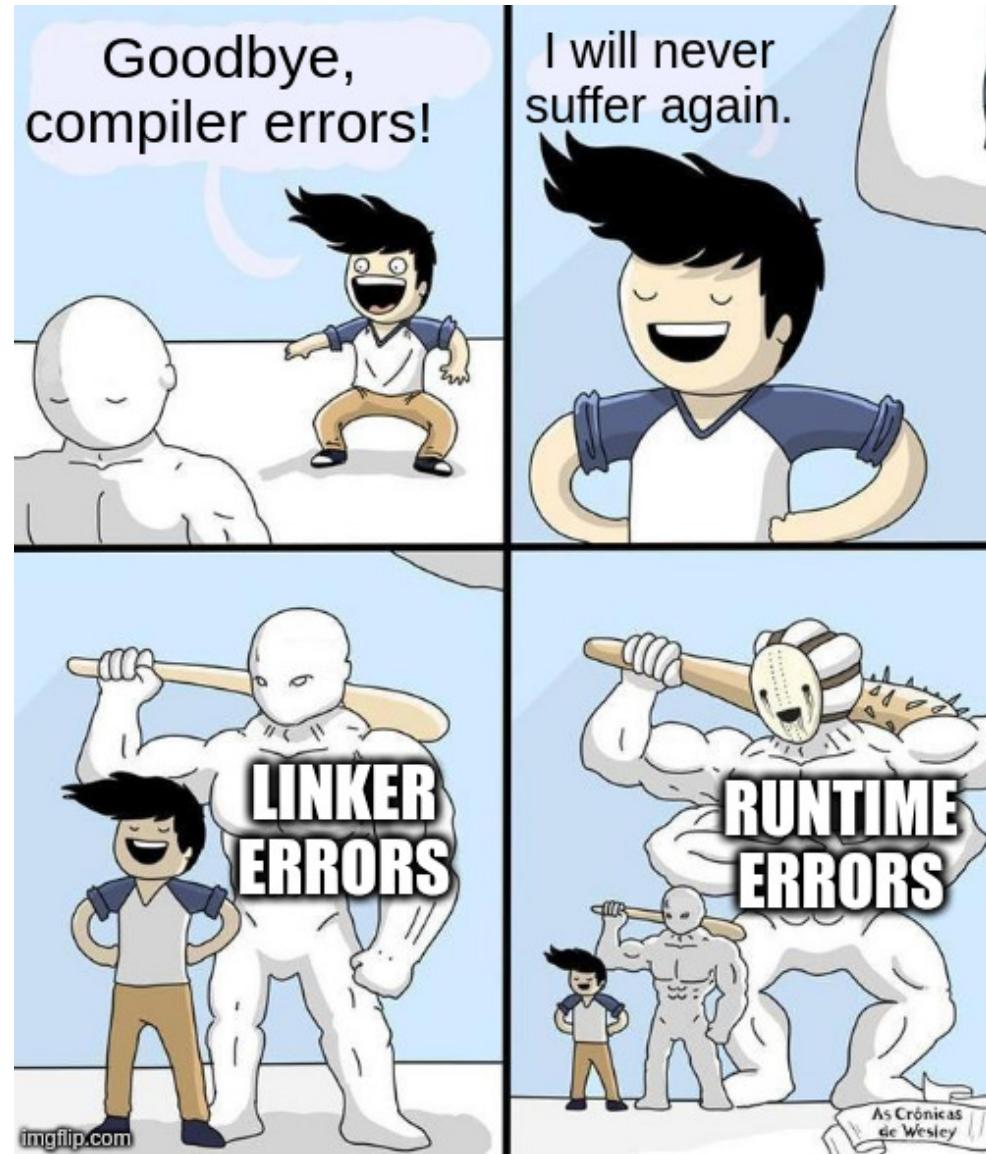


Figure 5: hatalar

Çalışma zamanı ve mantıksal hataları bulma

Yazdırma ifadeleri

İnsanlık programlamayı öğrendiği günden beri kodlarını hata ayıklamak için yazdırma ifadelerini kullanıyor.

Bu, kodunuzu hata ayıklamanın en zarif olmayan yolu olabilir, ancak küçük programları hata ayıklamanın en etkili yoludur.

Aşağıdaki kodu inceleyin.

```
class punchcard {
    public static void main(String[] args) {
        int a = 5 + 7; // HATA
        int b = 10;
        int c = a + b;
        if (c == 15) {
            System.out.println("Başarılı!");
        }
        else {
            System.out.println("Başarısız!");
        }
    }
}
```

Diyelim ki `a`'nın değerinin 5 ve `b`'nin değerinin 10 olmasını bekliyorduk. Sonuç olarak, `c`'nin değerinin 15

olmasını bekliyorduk. Ancak bir nedenden dolayı **c**'nin değeri 15 değil.

Yapabileceğimiz ilk şey, **c**'nin değerini görmek için **c**'nin değerini yazdırırmak.

```
class punchcard {  
    public static void main(String[] args) {  
        int a = 5 + 7; // HATA  
        int b = 10;  
        int c = a + b;  
        System.out.println(c); // HATA  
        if (c == 15) {  
            System.out.println("Başarılı!");  
        }  
        else {  
            System.out.println("Başarısız!");  
        }  
    }  
}
```

Çıktı:

22

Başarısız!

Artık **c** değerinin 22 olduğunu biliyoruz. Ancak **c** değerinin neden 22 olduğunu hâlâ bilmiyoruz.

a ve **b** değerlerini görüntüleyerek **a** ve **b** değerlerinin ne olduğunu görelim.

```
class punchcard {  
    public static void main(String[] args) {  
        int a = 5 + 7; // HATA  
        System.out.println(a); // HATA  
        int b = 10;  
        System.out.println(b); // HATA  
        int c = a + b;  
        System.out.println(c); // HATA  
        if (c == 15) {  
            System.out.println("Başarılı!");  
        }  
        else {  
            System.out.println("Başarısız!");  
        }  
    }  
}
```

Cıktı:

```
12  
10  
22  
Başarısız!
```

Anladık! **a** değeri 12 olduğu için **c** değeri 22. Bu hatayı düzeltebiliriz, **a** değerini 5'e değiştirerek.

Hatayı düzelttikten sonra, kodunuzu temiz tutmak için `System.out.println()` gibi yazdırma ifadelerini kaldırmayı unutmayın.

```
class punchcard {  
    public static void main(String[] args) {  
        int a = 5;  
        int b = 10;  
        int c = a + b;  
        if (c == 15) {  
            System.out.println("Başarılı!");  
        }  
        else {  
            System.out.println("Başarısız!");  
        }  
    }  
}
```

Çıktı:

Başarılı!

Birim test veya test durumları

Eğer girdiler basit değilse veya tüm kenar durumları bilmiyorsanız ne yapacaksınız?

Bu durumda, kodunuzu test etmek için birim testleri kullanabilirsiniz. Birim testleri, kodunuzu test etmenin bir yoludur, kod yazarak.

Ancak, birim testi konusu bu öğretici kapsamının dışındadır. Ayrıca, birim testleri Java'nın daha fazla bilgisini gerektirir.

Birim testi, küçük bir kodu hata ayıklamak için tercih edilmez.

Birim testleri, sadece geliştiriciye beklenmeyen bir davranışın meydana geldiğini bildirir. Sorunun nedeni hakkında bilgi vermez.

Ancak test durumları, birden fazla geliştirici tarafından geliştirilen büyük projelerin önemli bir parçasıdır.

Kodun beklenildiği gibi çalıştığından emin olur.



Figure 6: birim test
24

Hata Ayıklayıcılar

İşte sizi saatlerce hata ayıklamaktan kurtarabilecek en güçlü araç.

Hata ayıklayıcılar, kodunuzu satır satır çalıştırmanıza izin veren araçlardır.

Her kod satırındaki değişken değerini görebilirsiniz.

Adı sizi aldatmasın. Hata ayıklayıcılar sadece hata ayıklamak için kullanılmaz. Hata ayıklayıcılar harika:

1. **Yabancı bir kodun nasıl çalıştığını anlama.** Kodu satır satır çalıştırabilir ve her kod satırının ne yaptığını görebilirsiniz. Değerlerin nasıl değiştigini gözlemleyebilirsiniz.
2. **Bir hatanın nedenini bulma.** Kodu satır satır çalıştırabilir ve kodun beklenildiği gibi çalışmadığı yeri görebilirsiniz. Hangi değerin beklenildiği gibi olmadığını görebilirsiniz.
3. **Kodunuzu test etme.** Kodu satır satır çalıştırabilir ve kodun işe yarayıp yaramadığını görebilirsiniz. Kodun doğru sonucu verdiği ancak problemi çözmenin doğru yol olmadığı zamanlar vardır. Hata ayıklayıcılar bu tür problemleri bulmanıza yardımcı olabilir.

Hata ayıklayıcılarının temelleri

Öncelikle, hata ayıklayıcıların vazgeçilmez unsuru olan **kesme noktalarını (breakpoints)** öğrenelim.

- **Kesme Noktaları**

Kesme noktaları, kodunuzun yürütülmesini durdurmak istediğiniz noktalardır.

Bir kesme noktası belirlemek için satır numarasının sol tarafına tıklayarak belirleyebilirsiniz. Şekil 6.

Bu örneğe bakın.

The screenshot shows the Visual Studio Code interface with the following components:

- EXPLORER**: Shows a tree view of files under a folder named "TEST". The files listed are ".vscode", "digit.displayer.java", "digit.displldyer.2.java", "donut.java", "test.java" (selected), and "test2.java".
- EDITOR**: Displays the Java code for "test.java". The code defines a class named "punchcard" with a main method. It contains a bug where it adds 5 and 7 instead of 5 and 10. The terminal output shows the program running and printing "Failure!" because the condition in the if statement is false.
- TERMINAL**: Shows the command-line output of running the Java application. The output includes the command used to run the application, the Java version, and the result "Failure!".

```
1 class punchcard {
2     public static void main(String[] args) {
3         int a = 5 + 7; // BUG
4         int b = 10;
5         int c = a + b;
6         if (c == 15) {
7             System.out.println("Success!");
8         } else {
9             System.out.println("Failure!");
10    }
11 }
```

```
[suren@HaghDazhDefoov ~/Templates/Java/test]$ /usr/bin/env /usr/lib/jvm/java-21-openjdk/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /home/suren/.config/VS Codium/User/workspaceStorage/cf11f2490440d335edf9d9951da113b9/redhat.java/jdt_ws/test_c9889b72/bin punchcard
Picked up _JAVA_OPTIONS: -Djava.util.prefs.userRoot=/home/suren/.config/java
Failure!
[suren@HaghDazhDefoov ~/Templates/Java/test]$
```

Figure 7: kesme noktasi

Örnek:

```
class punchcard {  
    public static void main(String[] args) {  
        int a = 5 + 7; // HATA  
        int b = 10;  
        int c = a + b;  
        if (c == 15) {  
            System.out.println("Başarılı!");  
        }  
        else {  
            System.out.println("Başarısız!");  
        }  
    }  
}
```

Diyelim ki, **if** ifadesinden önce **a** ve **b** değerlerini görmek istiyorsunuz. 5. satıra bir kesme noktası belirleyebilir ve **hata ayıklama (debug)** düğmesine tıklayabilirsiniz. Şekil 8.

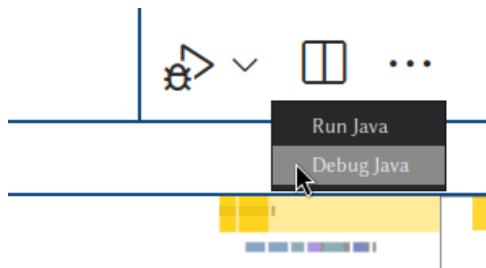


Figure 8: hata ayıklama düğmesi

The screenshot shows the Java debugger interface within an IDE. The top bar includes 'RUN A...', 'No Config' dropdown, and various icons. The title bar displays 'J test.java 4'. The main area shows the code for 'punchcard' class:

```
1 class punchcard {  
2     public static void main(String[] args) { args = String  
3         int a = 5 + 7; // BUG a = 12  
4         int b = 10; b = 10  
5         int c = a + b; a = 12, b = 10  
6         if (c == 15) {  
7             System.out.println("Success!");  
8         }  
9         else {  
10            System.out.println("Failure!");  
11        }  
12    }  
13 }  
14
```

The line 'a = 12, b = 10' is highlighted in yellow, indicating a bug. The variable 'args' is listed in the 'VARIABLES' panel under 'Local' with value 'String[0]@8'. The 'WATCH' and 'CALL STACK' panels are empty. The 'BREAKPOINTS' panel shows a checked checkbox for 'test.java'. The bottom navigation bar includes 'PROBLEMS' (16), 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (selected), and 'PORTS'. The terminal window shows:

```
t_ws/test_c9889b72/bin punchcard  
Picked up _JAVA_OPTIONS: -Djava.util.prefs.userRoot=/home/sure  
n/.config/java
```

A context menu is open on the right side of the terminal window, listing options: 'Java Build St...', 'Run: punchc...', and 'Debug: punc...'. A circular icon with the number '5' is located at the bottom center.

Figure 9: hata ayıklayıcı

Bu bir hata ayıklama oturumu. Şekil 9.

Hata ayıklayıcının birçok kullanışlı aracı vardır. Bunları tek tek gözden geçirelim.

Ancak öncelikle, bir Java programının akışını anlayalım.

Şimdilik, '

`public static void main(String[] args)` içinde, kodunuzun üstten alta doğru satır satır çalıştığını bilmemiz yeterli.



Figure 10: adım göstergesi

Adım Göstergesi Bu kahverengi sekizgen-ok işaretli düğme, **adım göstergesi**dir. Şekil 10.

Bu düğme, bir sonraki yürütülecek kod satırını gösterir. Kod, adım göstergesinde gösterilen satırda durur.

Şimdi, `c = a + b;` ifadesinden önceki tüm bilgileri görebiliriz.

Değişkenler Akışın durduğu noktadan önce değişken değerlerini görmek için birden fazla yol vardır. Şekil 11.

✓ VARIABLES

✓ Local

args: String[0]@8
a: 12
b: 10

```
J test.java > punchcard > main(String[])
1 chcard {
Run | Debug
2 public static void main(String[] args) { args = String[0]@8
3     int a = 5 + 7; // BUG a = 12
4     int b = 10; b = 10
5     int c = a + b; a = 12, b = 10
6     if (c == 15) {
7
8     }
9 }
```

Figure 11: değişkenler

- Değişkenler Paneli

Solda, değişkenler panelini görebilirsiniz. Akışın durduğu noktadan önceki tüm değerler bu panoda gösterilir.

- Üzerine gelme

Değerinizi görmek için değişkenin üzerine gelip değeri görebilirsiniz. 3. satırdaki **a** üzerine gelerek **12** değerini görebilirsiniz.

- İçinde İpucu (Inlay hints)

Değişkenlerin değerleri, gerçek kodun hemen sonrasında gri renkli metin olarak gösterilir.

- Hata Ayıklama Konsolu

Değişkenin değerini hata ayıklama konsolunda çağırabilirsiniz. Şekil 12.

Bir değişkenin değerini görmek için **System.out.println()** kullanmaya gerek yoktur. Sadece hata ayıklama konsolunda değişken adını yazın ve enter tuşuna basın.

PROBLEMS

16

OUTPUT

DEBUG CONSOLE

TERMINAL

→ a

12

→ b

10

→ a+b - 7

15

> a+b

Figure 12: hata ayıklama konsolu

Hata Ayıklama Araç Çubuğu Şimdi, 10. satıra bir kesme noktası ekleyelim. Bu ikinci kesme noktasını bu bölümde daha sonra kullanacağımız.



Figure 13: hata ayıklama araç çubuğu

Bu, hata ayıklama araç çubuğudur. Şekil 13.

Soldan sağa:

- **Devam Et (Continue)**

Bu düğme, bir sonraki kesme noktasına kadar kodun yürütülmesini sağlar. Unutmayın, adım göstergesi tarafından belirtilen satırda durur, bu yüzden `System.out.println("Başarısız!")` henüz çalıştırılmadı. Şekil 14.

- **Üzerinden Atlama (Step Over)**

Bu düğme, kodun bir sonraki satırında yürütme akışını durdurur. Bir sonraki satırda bir kesme noktası olsa da olmasa da fark etmez.

Örneğin, 3. satıra bir kesme noktası koyabilir ve **üzerinden atlama** düğmesine tıklayabilirsiniz. Yürütme akışı, 4. satırda durur. Bir sonraki düğme basımı, yürütme akışını 5. satırda durdurur. Şekil 15.

- **İçine Gir (Step Into)**

The screenshot shows the VS Code interface with the Java extension loaded. The top bar includes tabs for 'RUN A...', 'No Config', and '...'. The title bar shows the file 'test.java' with line 4 selected. The main editor area displays the following Java code:

```
1 class punchcard {  
2     public static void main(String[] args) { args = String  
3         int a = 5 + 7; // BUG a = 12  
4         int b = 10; b = 10  
5         int c = a + b; c = 22, a = 12, b = 10  
6         if (c == 15) { c = 22  
7             System.out.println("Success!");  
8         }  
9         else {  
10            System.out.println("Failure!");  
11        }  
12    }  
13 }
```

The code editor has several annotations: 'Run | Debug' is shown above line 1; 'args = String' is highlighted in yellow on line 3; 'a = 12' is highlighted in yellow on line 4; 'c = 22, a = 12, b = 10' is highlighted in yellow on line 5; 'c = 22' is highlighted in yellow on line 6; 'System.out.println("Success!");' is highlighted in yellow on line 7; 'System.out.println("Failure!");' is highlighted in yellow on line 10; and the brace for line 11 is highlighted in yellow.

The left sidebar contains the 'VARIABLES' and 'WATCH' toolbars. The 'VARIABLES' toolbar shows local variables: 'args: String[0]@8', 'a: 12', 'b: 10', and 'c: 22'. The 'WATCH' toolbar has a single entry: '10' with the value 'System.out.println("Failure!");'.

The bottom navigation bar includes 'PROBLEMS' (16), 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (selected), and 'PORTS'. The 'TERMINAL' tab shows the command-line output:

```
=y,address=localhost:44301 --enable-preview -XX:+ShowCodeDetail  
lsInExceptionMessages -cp /home/suren/.config/VSCode/User/wor  
kspaceStorage/cf11f2490440d335edf9d9951da113b9/redhat.java/jd  
t_ws/test_c9889b72/bin punchcard  
Picked up _JAVA_OPTIONS: -Djava.util.prefs.userRoot=/home/sure  
n/.config/java
```

The right sidebar shows the 'Java Build St...', 'Run: punchc...', and 'Debug: punc...' options.

Figure 14: devam et

The screenshot shows a Java code editor interface. The top bar displays the file name "J test.java" and a line count of "4". To the right is a toolbar with various icons: a dropdown menu, a play button, a refresh button, a save button, an up arrow, a down arrow, a circular arrow, a square with a diagonal line, and a lightning bolt icon.

The code editor window shows the following Java code:

```
1 class punchcard {
2     public static void main(String[] args) {
3         int a = 5 + 7; // BUG a = 12
4         int b = 10; b = 10
5         int c = a + b; a = 12, b = 10
6         if (c == 15) {
```

The word "punchcard" is underlined with a wavy yellow line, indicating a potential misspelling or error. The variable "a" in the first assignment statement is highlighted in yellow, and the value "12" is also highlighted in yellow, suggesting a bug or highlighting. The variable "b" in the second assignment statement is highlighted in yellow, and the value "10" is also highlighted in yellow. The variable "c" in the third assignment statement is highlighted in yellow. The entire line "int c = a + b; a = 12, b = 10" is highlighted in yellow, and the values "a = 12, b = 10" are also highlighted in yellow, further emphasizing the bug.

Figure 15: üzerinden atlama

Bu düğmenin eylemi, **üzerinden atlama** düğmesinin benzeridir. Farkı bu öğretici kapsamı dışındadır. (Yöntem hata ayıklamak için kullanışlıdır)

- **Dışına Çık (Step Out)**

Bu düğmenin eylemi bu öğretici kapsamı dışındadır. (Yöntem hata ayıklamak için kullanışlıdır)

- **Yeniden Başlat (Restart)**

Bu düğme, hata ayıklama oturumunu yeniden başlatır. Birden fazla kez kodu hata ayıklamak isterseniz veya kodu değiştirdiğiniz ve kodu tekrar hata ayıklamak istiyorsanız kullanışlıdır.

- **Durdur (Stop)**

Bu düğme, hata ayıklama oturumunu durdurur. Ayrıca hata ayıklama aracını keser, böylece IDE'nin normal moduna geri dönebilir ve kodu normal şekilde çalıştırabilirsiniz.

Kesme Noktaları Normalde, bir kesme noktasına artık ihtiyacınız yoksa, onu tıklayarak kaldırabilirsiniz, ancak bazen kesme noktasını korumak isteyebilirsiniz, ancak yürütme akışını durdurmak istemezsınız. (şimdilik)

Bu durumda, kesme noktasını devre dışı bırakabilirsiniz. Şekil 16.

✓ BREAKPOINTS



Uncaught Exceptions

Caught Exceptions

● ✓ test.java

3



Figure 16: kesme noktasını devre dışı bırak



Figure 17: Hata Ayıklayıcı Kullanmak?!

Gelişmiş Hata Ayıklama

Hata ayıklayıcıların birçok daha gelişmiş özelliği var. Ancak bu özellikler bu öğretici kapsamının dışındadır, Ancak burada bazlarını belirtiyorum.

Certainly! Here is the translation of the document into Turkish:

Koşullu kesme noktaları Normalde, hata ayıklayıcı istediğiniz noktada durur. Bir döngü içindeyseniz veya sorunu tetikleyen değerleri biliyorsanız, bunu istemezsiniz. Koşullu bir kesme noktası, kesme noktanıza biraz Java kodu eklemenizi sağlar, böylece yalnızca o koşul doğru olduğunda durur. Bu yaklaşım, önemsedığınız değere ulaşana kadar defalarca devam etmekten kaçınmanızı sağlar.

Bu özellik özellikle döngülerin hata ayıklanmasında kullanışlıdır.

İzleme Noktaları İzleme noktaları kesme noktalarına benzer. Ancak, yürütmenin akışını durdurmak yerine, değişkenin değerini yalnızca yazdırırlar.

Bu, print ifadelerini kullanmaktan daha iyidir çünkü hatayı düzelttikten sonra izleme noktasını kaldırmanız gerekmek.

Değişken Değerlerini Değiştirme Hata ayıklayıcıda değişken değerlerini manuel olarak değiştirebilir ve kodun, orijinal değer yerine yeni değerle devam etmesini sağlayabilirsiniz.

Bu, kodu değiştirmeden potansiyel düzeltmeleri test etmenize yardımcı olur.

Sonuç

Hata ayıklama, bir programcının en önemli becerilerinden biridir. Her gün kullanacağınız bir beceridir. Her projede kullanacağınız bir beceridir.

Öğrenin, Kullanın, Ustalaşın.

Ve her zamanki gibi, Mutlu kodlamalar!

Bu adam olmayın!

Spending 4 hours fixing a bug and realizing
that the code you've been compiling was
not the code you've been fixing



Figure 18: The guy

Bu adam olun!

Programmers while reviewing the codes

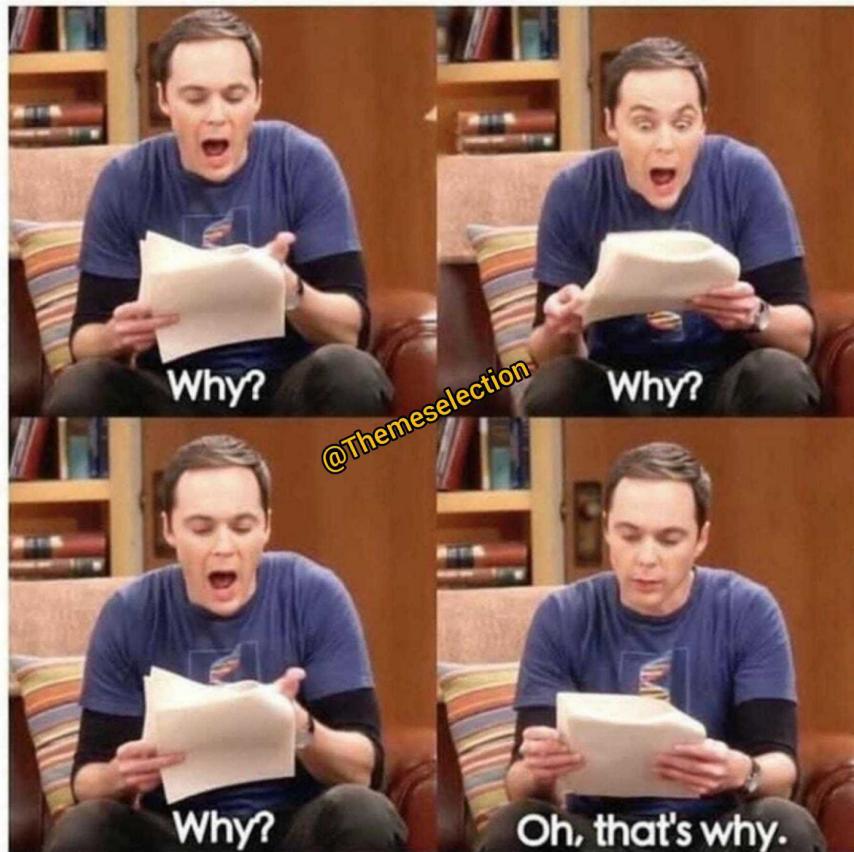


Figure 19: The other guy

Belgelendirme

Her IDE'nin kendi hata ayıklayıcısı bulunmaktadır. Bu nedenle hata ayıklayıcıların belgelendirmesi her IDE için farklıdır.

IDE'nizin belgelendirmesine giderek hata ayıklayıcılar hakkında daha fazla bilgi edinebilirsiniz.

- IntelliJ IDEA
- Eclipse
- NetBeans
- Visual Studio Code
- vim