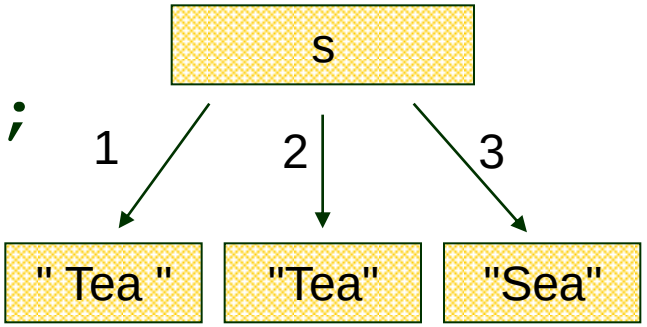
מחרוזות ב Java
ותכנות מונחה בדיקות
(Test Driven Development)

תרגול 8 – תוכנה 1

String Immutability

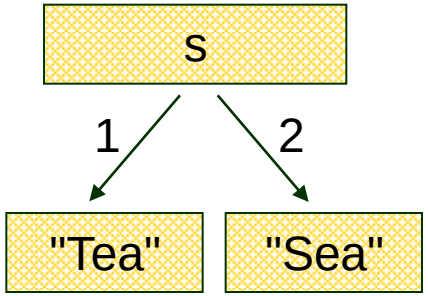
- Strings are constants

```
String s = " Tea ";
s = s.trim();
s = s.replace('T', 'S');
```



- A string reference may be set:

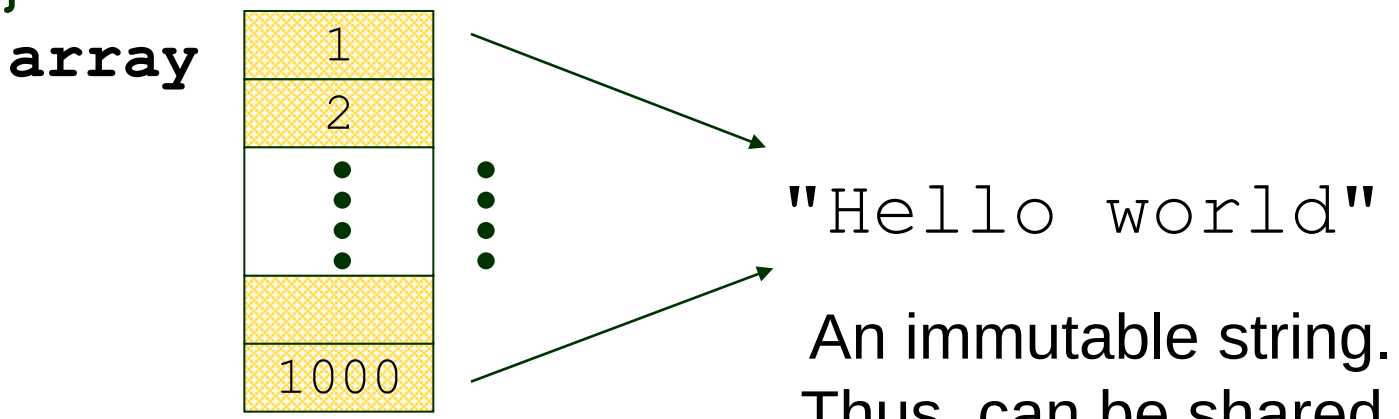
```
String s = "Tea";
s = "Sea";
```



String Interning

- Avoids duplicate strings

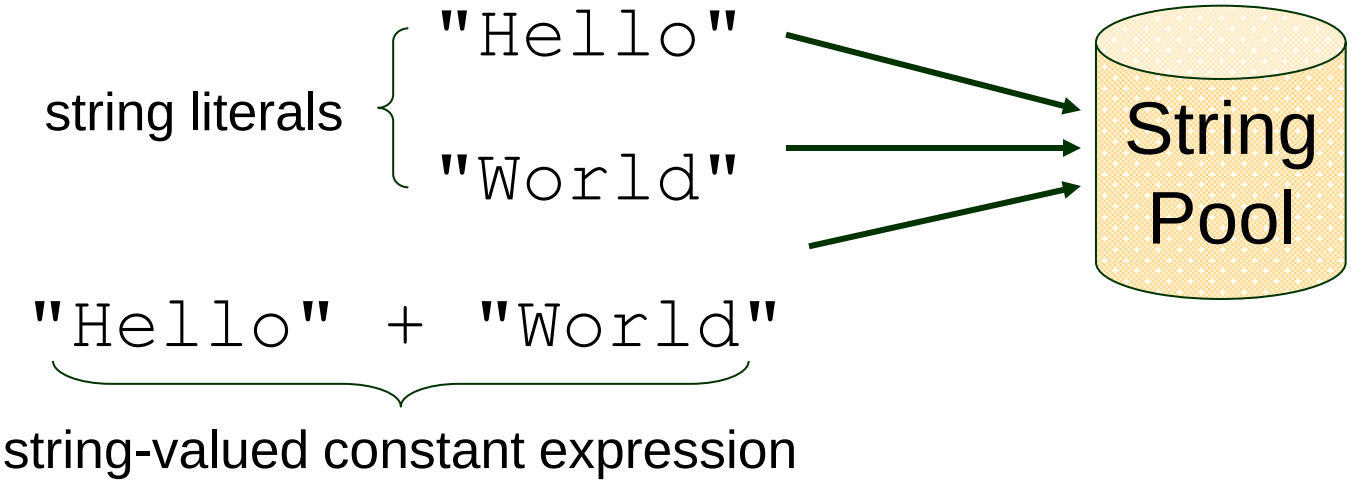
```
String[] array = new String[1000];
for (int i = 0; i < array.length; i++) {
    array[i] = "Hello world";
}
```



An immutable string.
Thus, can be shared.

String Interning (cont.)

- All string literals and string-valued constant expressions are interned.



String Interning (cont.)

- The `String` class has a static private **pool** of internal strings.
- `myString.intern()` implementation:

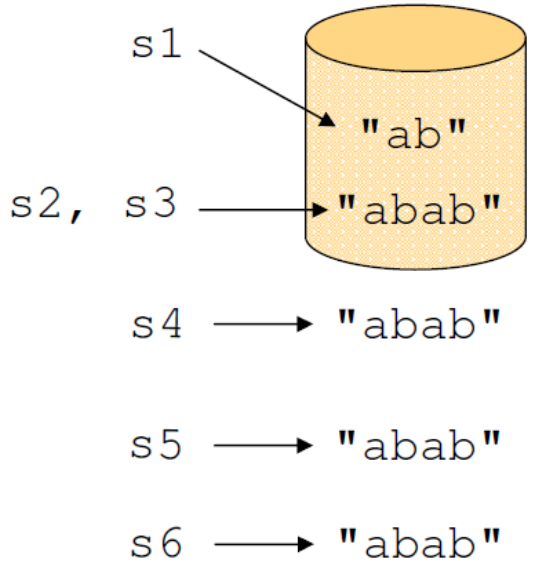
if $\exists s \in pool$: *myString.equals(s) == true*
return s;
else
add myString to the pool
return myString;

`equals`: compares characters
`==`: compares references

String Interning (cont.)

If:

```
String s1 = "ab";
String s2 = "ab" + "ab";
String s3 = "aba" + "b";
String s4 = s1 + s1;
String s5 = s1 + s1;
String s6 = s1 + "ab";
```



Then:

```
s4.equals(s2) is true
(s4 == s2) is false
(s4 == s5) is false
(s2 == s3) is true
(s2 == s6) is false
(s4.intern() == s2) is true
(s4.intern() == s5.intern()) is true
```

String Constructors

- Use implicit constructor:

```
String s = "Hello";
```

(string literals are interned)

Instead of:

```
String s = new String("Hello");
```

(causes extra memory allocation)

The StringBuilder Class

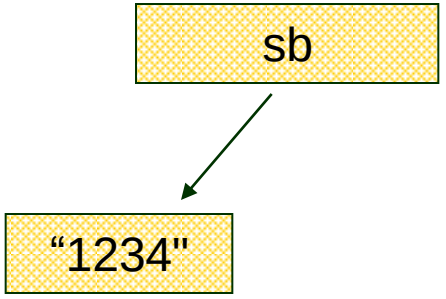
- Represents a **mutable** character string
- Main methods: **`append()` & `insert()`**
  - accept data of any type
  - If: **`sb = new StringBuilder("123")`**

    Then: **`sb.append(4)`**

    is equivalent to

    `sb.insert(sb.length(), 4)`

    Both yield **`"1234"`**



The Concatenation Operator (+)

- String conversion and concatenation:
  - `"Hello " + "World"` is `"Hello World"`
  - `"19" + 8 + 9` is `"1989"`

- Concatenation by `StringBuilder`
  - `int i = 8; String x = "19" + i + 9;`

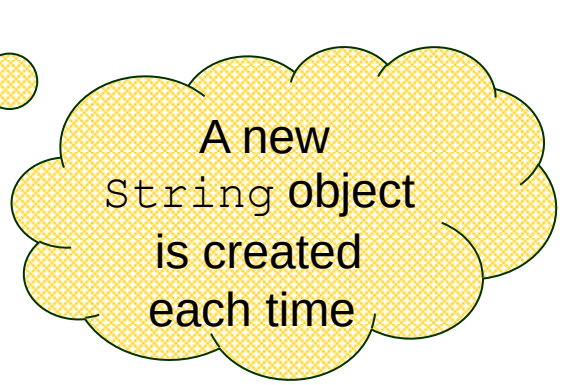
    is compiled to the equivalent of:

```
String x = new StringBuilder().append("19").
                    append(8).append(9).toString();
```

StringBuilder vs. String

- Inefficient version using String

```
public static String duplicate(String s, int times) {
    String result = s;
    for (int i = 1; i < times; i++) {
        result = result + s;
    }
    return result;
}
```



StringBuilder vs. String (cont.)

- More efficient version with StringBuilder:

```
public static String duplicate(String s, int times) {
    StringBuilder result = new StringBuilder(s);
    for (int i = 1; i < times; i++) {
        result.append(s);
    }
    return result.toString();
}
```

StringBuilder vs. String (cont.)

- Even more efficient version:

```
public static String duplicate(String s, int times) {
    StringBuilder result =
         new StringBuilder(s.length() * times);
    for (int i = 0; i < times; i++) {
        result.append(s);
    }
    return result.toString();
}
```

created with the correct capacity

StringBuilder vs. StringBuffer

- StringBuilder has the same API as StringBuffer, but with no guarantee of synchronization.
- StringBuilder is a replacement for StringBuffer when there is only a single thread
- Where possible, it is recommended to use StringBuilder as it will be faster under most implementations.

מבוסס על הספר:

Test-Driven Development By Example

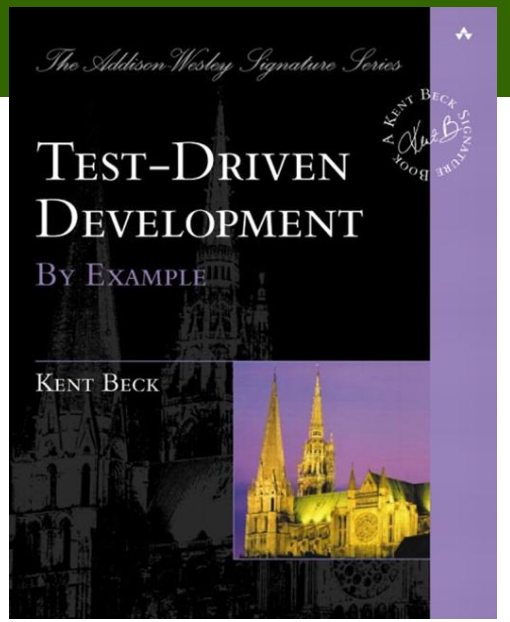
By Kent Beck

Publisher: Addison Wesley

Date: November 08, 2002

ISBN: 0-321-14653-0

Pages: 240



בדיקות תחילה

- קוד נקי שעובד (clean code that works)
  - קוד חדש נכתב רק אחרי שבדיקה אוטומטית נכשלת
  - הסרת כפילויות
- השלכות טכניות
  - התכן (design) מלווה בקידוד
  - המתכנת כותב את הבדיקות
  - קומפילציה מהירה
  - צימוד חלש בין רכיבים

אדום – ירוק - שכתוב

- אדום
  - כתוב בדיקה שנכשלת (אולי אפילו לא עוברת קומפילציה)
- ירוק
  - תעשה במהירות שהבדיקה תצליח (תוך אולי שחיטת פרות קדושות של עקרונות תכנות נכונים)
- שכתוב (refactoring)
  - הסר את הכפילויות בקוד שכראה הכנסת בשלב הקודם

Fibonacci

- דוגמא פשוטה
- ברצוננו לכתוב פונקציה המחשבת איבר בסדרת פיבונאצ'י
- נכתוב את הפונקציה בגישה של Test First

שלבי העבודה

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change.
4. Run all tests and see them all succeed.
5. Refactor to remove duplication.

איך מתחילים?

- התכנות שלנו מונע מסיפורים (תסריטים)
  - "נרצה שאפשר יהיה לבצע במערכת..."
- מה נרצה שתעשה הפונקציה?
- ניצור מחלקה ששם יישב קוד הבדיקה

```
public class TestFib extends TestCase {
   public void testFibonacci() {
          assertEquals(0, fib(0));
   }
}
```

TODO List:
fib(0) == 0

JUnit Anotations

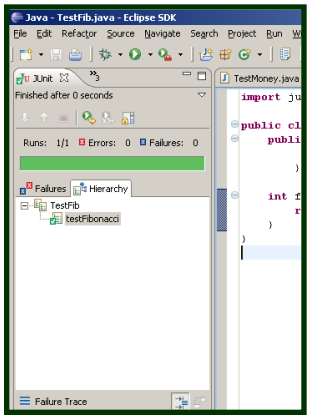
- @Test
- @Before , @After
- @BeforeClass , @AfterClass
- @Ignore
- @Test(timeout=100)

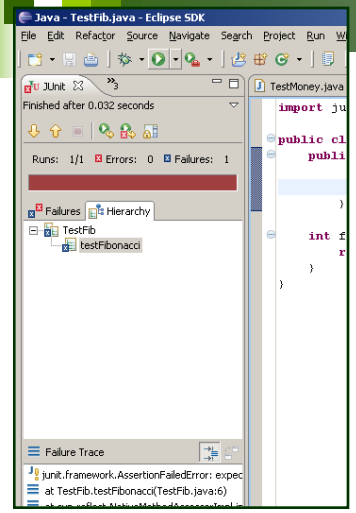
תעשה שיתקמפל

- נוסיף קוד מינימלי כדי לפתור את בעיית הקומפילציה

```
int fib(int i){
        return 0;
}
```

- נריץ... (את קוד הבדיקה)
- ירוק





נוסיף עוד בדיקה

- אפשר להוסיף עוד מתודת בדיקה חדשה:

```
public void testFibonacciOfOneIsOne() {
   assertEquals(1, fib(1));
}
```

- אנחנו נסתפק ב:

```
public void testFibonacci() {
   assertEquals(0, fib(0));
   assertEquals(1, fib(1));
}
```

- נריץ... אדום

TODO List:

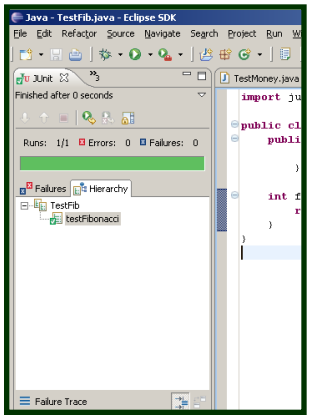
~~fib(0) == 0~~

fib(1) == 1

תעשה שיהיה ירוק

- נוסיף קוד מינימלי כדי להפוך את הפס לירוק

```
int fib(int n) {
   if (n == 0)
       return 0;
   return 1;
}
```



- נריץ... (את קוד הבדיקה)
- ירוק

TODO List:

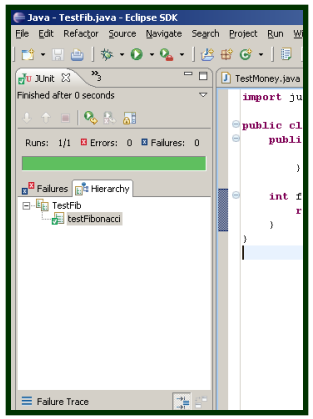
~~fib(0) == 0~~

fib(1) == 1

הסרת כפילויות

- הכפילויות הפעם הן בבדיקה (ולא בקוד) – נסיר אותן (refactoring)

```
public void testFibonacci() {
   int cases[][]= {{0,0},{1,1}};
   for (int i= 0; i < cases.length; i++)
       assertEquals(cases[i][1], fib(cases[i][0]));
}
```



- נוודא שלא הרסנו כלום (או שלא גילינו באג חדש)...

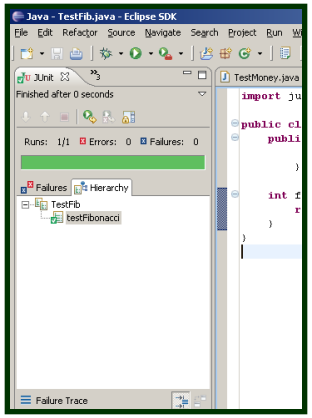
ירוק

נוסיף עוד בדיקה

- קל להוסיף את הבדיקה בפונקציה הבדיקה המשוכתבת (6 הקשות מקלדת בלבד!)

```
public void testFibonacci() {
   int cases[][]= {{0,0},{1,1},{2,1}};
   for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

- נריץ...
- עדיין ירוק



TODO List:

~~fib(0) == 0~~

~~fib(1) == 1~~

fib(2) == 1

ואחת לשנה הבאה

- רק בשביל להיות בטוחים שסיימנו...

```
public void testFibonacci() {
   int cases[][]= {{0,0},{1,1},{2,1},{3,2}};
   for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```

- נריץ... אדום (שזה טוב, גילינו באג!)

TODO List:

~~fib(0) == 0~~

~~fib(1) == 1~~

~~fib(2) == 1~~

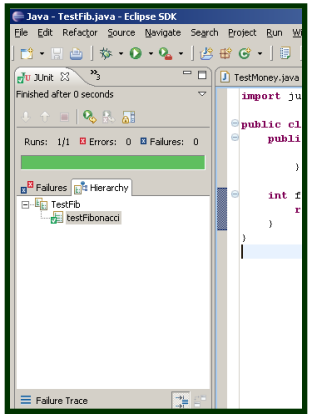
fib(3) == 2

תעשה שיהיה ירוק

- נוסיף קוד מינימלי כדי להפוך את הפס לירוק

```
int fib(int n) {
   if (n == 0) return 0;
   if (n <= 2) return 1;
   return 2;
}
```

שכתוב:
מאיפה הגיע ה-2 ?
זהו בעצם 1+1



- נריץ... ירוק

TODO List:

~~fib(0) == 0~~

~~fib(1) == 1~~

~~fib(2) == 1~~

fib(3) == 2

שכתוב

- קיבלנו:

```
int fib(int n) {
   if (n == 0) return 0;
   if (n <= 2) return 1;
   return 1 + 1;
}
```

- ה-1 הראשון הוא בעצם fib(n-1)

```
int fib(int n) {
   if (n == 0) return 0;
   if (n <= 2) return 1;
   return fib(n-1) + 1;
}
```

- ה-1 השני הוא בעצם fib(n-2)

שכתוב

- ה-1 השני הוא בעצם fib(n-2)

```
int fib(int n) {
   if (n == 0) return 0;
   if (n <= 2) return 1;
   return fib(n-1) + fib(n-2);
}
```

- נכליל עבור fib(2), וסיימנו

```
int fib(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   return fib(n-1) + fib(n-2);
}
```

סיכום

- Made a list of the tests we knew we needed to have working
- Told a story with a snippet of code about how we wanted to view one operation
- Made the test compile with stubs
- Made the test run by committing horrible sins
- Gradually generalized the working code, replacing constants with variables
- Added items to our to-do list rather than addressing them all at once

בהקשר כללי יותר

- Write a test.
- Make it run
- Make it right

Write a test

- Think about how you would like the operation in your mind to appear in your code.
- You are writing a story. Invent the interface you wish you had.
- Include all of the elements in the story that you imagine will be necessary to calculate the right answers.

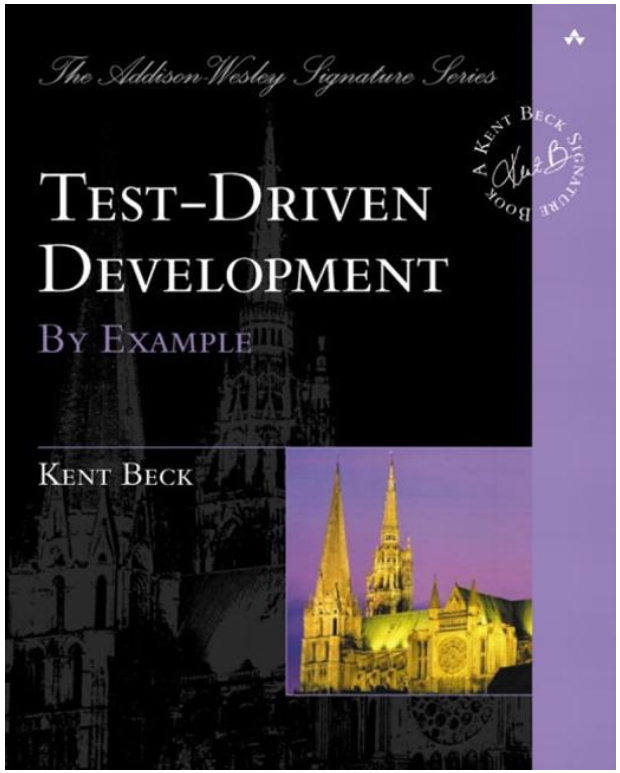
Make it run

- Quickly getting that bar to go to green dominates everything else
- If a clean, simple solution is obvious, then type it in
- If the clean, simple solution is obvious but it will take you a minute, then make a note of it and get back to the main problem, which is getting the bar green in seconds
- This shift in aesthetics is hard for some experienced software engineers
- They only know how to follow the rules of good engineering
- Quick green excuses all sins. But only for a moment

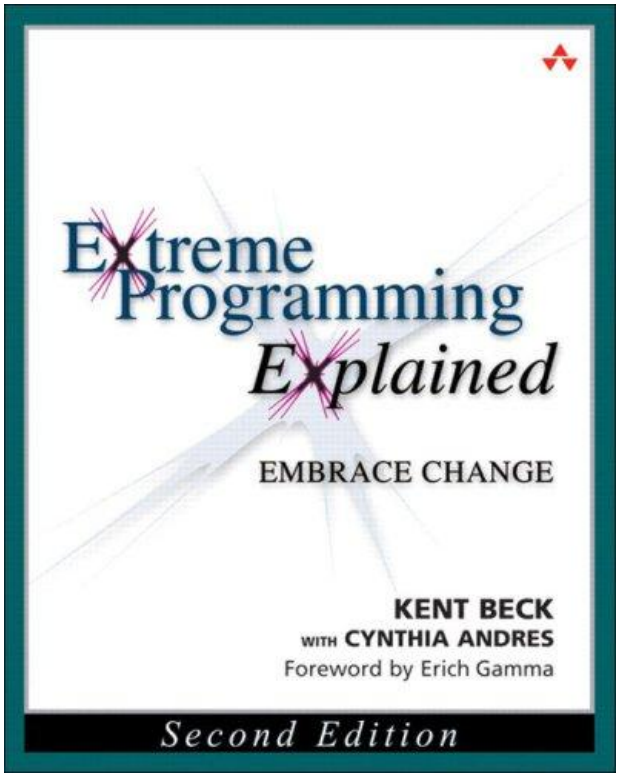
Make it right

- Now that the system is behaving, put the sinful ways of the recent past behind you
- Step back onto the straight and narrow path of software righteousness
- Remove the duplication that you have introduced, and get to green quickly

רוצים עוד?



עוד TDD



עוד XP