תרגול מספר 3

העמסה ומחרוזות

מה בתוכנית

- העמסת פונקציות והנדסת תוכנה
  - נוחות, תאימות לאחור ושכפול קוד
- שימוש במחלקות קיימות:
  - String ו- StringBuffer

העמסת פונקציות (שיקולי נוחות)

- נממש את max המקבלת שני ארגומנטים ומחזירה את הגדול מביניהם
- ללא שימוש בתכונת ההעמסה (בשפת C למשל) היה צורך להמציא שם נפרד עבור כל אחת מהמתודות:

```
public class MyUtils {
    public static double max_double(double x, double y)
    { ... }

    public static long max_long(long x, long y)
    { ... }
}
```

- השמות מלאכותיים ולא נוחים

העמסת פונקציות (פחות נוחות)

- בעזרת מנגנון ההעמסה ניתן להגדיר:
  - `public static double max(double x, double y)`
  - `public static long max(long x, long y)`
- בחלק מהמקרים, אנו משלמים על הנוחות הזו
- למשל, איזו מהפונקציות תופעל במקרים הבאים:

```
max(1L , 1L);          // max(long,long)
max(1.0 , 1.0);        // max(double,double)
max(1L , 1.0);         // max(double,double)
max(1 , 1);            // max(long,long)
```

העמסה והקומפיילר

- המהדר מנסה למצוא את הגרסה **המתאימה ביותר** עבור כל קריאה לפונקציה על פי טיפוסי הארגומנטים של הקריאה
- אם אין התאמה מדויקת לאף אחת מחתימות השירותים הקיימים, המהדר מנסה המרות (casting) שאינן מאבדות מידע
- אם לא נמצאת התאמה או שנמצאות שתי התאמות "באותה רמת סבירות" או שפרמטרים שונים מתאימים לפונקציות שונות המהדר מודיע על אי בהירות (ambiguity)

העמסה וערכי ברירת מחדל לארגומנטים

- נתאר מצב שבו פונקציה המקבלת מספר ארגומנטים, מרבה לקבל את אותם ערכים עבור חלק מהארגומנטים

- כדי לחסוך ממי שקורא לפונקציה את הצורך לציין את כל הארגומנטים (אולי הם רבים, אולי ערכיהם איזוטרים), נעמיס גרסה שבה יש לציין רק את הארגומנטים "החשובים באמת"

העמסה וערכי ברירת מחדל לארגומנטים

- לדוגמא:
  - פונקציה לביצוע ניסוי פיזיקלי מורכב:

```
public static void doExperiment(double altitude,
                double presure, double volume, double mass){
    ...
}
```

  - אם ברוב הניסויים המתבצעים הגובה, הלחץ והנפח אינם משתנים, יהיה זה מסורבל לציין אותם בכל קריאה לפונקציה

  - לשם כך, נגדיר עוד גרסה יותר קומפקטית המקבלת כארגומנט רק את המסה:

```
public static void doExperiment(double mass){
    ...
}
```

העמסה ותאימות לאחור

- נניח כי כתבת את הפונקציה השימושית :

```
public static int compute(int x)
```

המבצעת חישוב כלשהו על הארגומנט **x**

- לאחר זמן מה, כאשר הקוד כבר בשימוש במערכת (גם מתוך מחלקות אחרות שלא אתה כתבת), עלה הצורך לבצע חישוב זה גם בבסיסי ספירה אחרים (החישוב המקורי בוצע בבסיס עשרוני)

- בשלב זה לא ניתן **להחליף** את חתימת הפונקציה להיות:

```
public static int compute(int x, int base)
```

מכיוון שקטעי הקוד המשתמשים בפונקציה יפסיקו להתקמפל

העמסה, תאימות לאחור ושכפול קוד

- על כן במקום **להחליף** את חתימת השרות **נוסיף** פונקציה חדשה כגירסה מועמסת
  - משתמשי הפונקציה המקורית לא נפגעים
  - משתמשים חדשים יכולים לבחור האם לקרוא לפונקציה המקורית או לגרסה החדשה ע"י העברת מספר ארגומנטים מתאים
- בעיה – קיים דמיון רב בין המימושים של הגרסאות המועמסות השונות (גוף המתודות compute)
- דמיון רב מדי - שכפול קוד זה הינו בעייתי מכמה סיבות שנציין מיד

שכפול קוד
הוא הדבר הנורא ביותר
בעולם (התוכנה) !

העמסה, שכפול קוד ועקביות

- חסרונות שכפול קוד:
  - קוד שמופיע פעמיים, יש לתחזק פעמיים – כל שינוי, שדרוג או תיקון עתידי יש לבצע בצורה עקבית בכל המקומות שבהם מופיע אותו קטע הקוד
- כדי לשמור על עקביות שתי הגרסאות של **compute** נממש את הגרסה הישנה בעזרת הגרסה החדשה:

```
public static int compute(int x, int base){
   // complex calcultion...
}

public static int compute(int x){
    return compute(x, 10);
}
```

העמסת מספר כלשהו של ארגומנטים

- ב Java5 התווסף תחביר להגדרת שרות עם **מספר לא ידוע** של ארגומנטים (vararg)
- נניח שברצוננו לכתוב פונקציה שמחזירה את ממוצע הארגומנטים שקיבלה:

```
public static double average(double x){
   return x;
}

public static double average(double x1, double x2){
   return (x1 + x2) / 2;
}

public static double average(double x1, double x2, double x3){
   return (x1 + x2 + x3) / 3;
}
```

- למימוש 2 חסרונות:
  - שכפול קוד
  - לא תומך בממוצע של 4 ארגומנטים

העמסת מספר כלשהו של ארגומנטים

- רעיון: הארגומנטים יועברו כמערך

```
public static double average(double [] args) {
   double sum = 0.0;
   for (double d : args) {
      sum += d;
   }
   return sum / args.length;
}
```

- **יתרון**: שכפול הקוד נפתר
- **חסרון**: הכבדה על הלקוח - כדי לקרוא לפונקציה יש ליצור מערך

```
public static void main(String[] args) {
   double [] arr = {1.0, 2.0, 3.0};
   System.out.println("Averge is:" + average(arr));
```

ג'אווה באה לעזרה

- תחביר מיוחד של שלוש נקודות (...) יוצר את המערך מאחורי הקלעים:

```
public static double average(double ... args) {
   double sum = 0.0;
   for (double d : args) {
      sum += d;
   }
   return sum / args.length;
}
```

- ניתן כעת להעביר מפונקציה מערך או ארגומנטים בודדים:

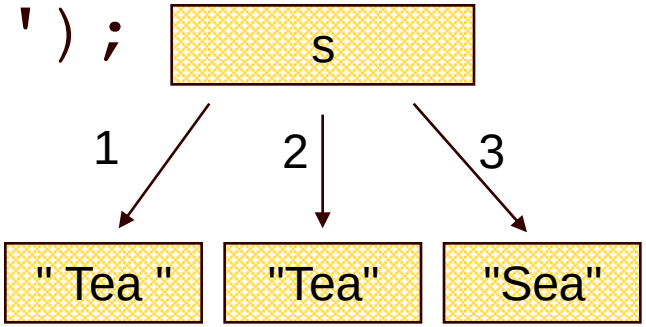
```
double [] arr = {1.0, 2.0, 3.0};
System.out.println("Averge is:" + average(arr));
System.out.println("Averge is:" + average(1.0, 2.0, 3.0));
```

מחרוזות

String Immutability

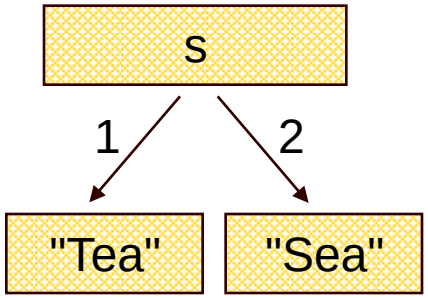
- Strings are constants

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



- A string reference may be set:

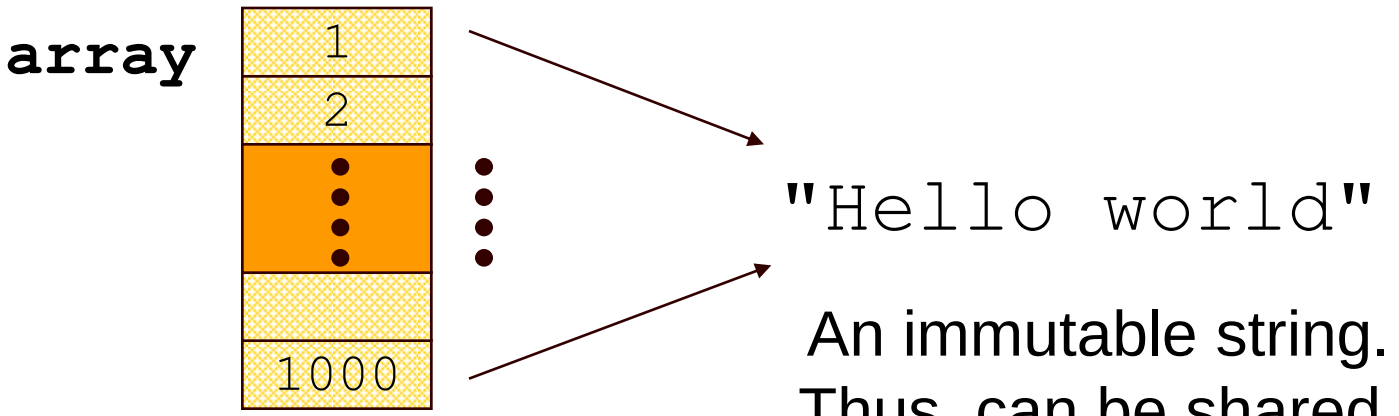
```
String s = "Tea";
s = "Sea";
```



String Interning

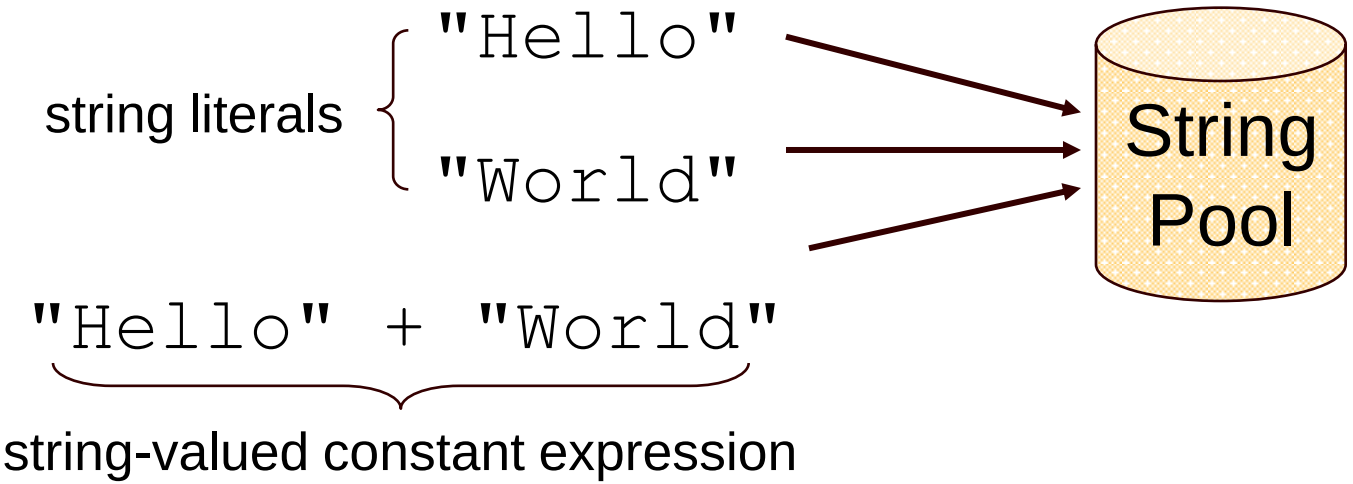
- Avoids duplicate strings

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



String Interning (cont.)

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



String Constructors

- Use implicit constructor:

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

(string literals are interned)

Instead of:

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

(causes extra memory allocation)

The StringBuffer Class

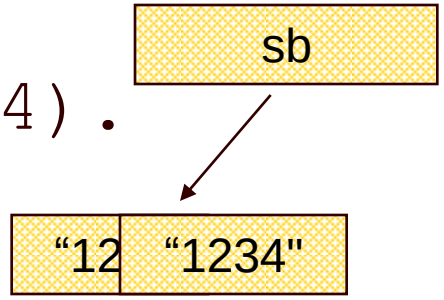
- Represents a **mutable** character string
- Main methods: `append()` & `insert()`
  - accept data of any type
  - If: `sb = new StringBuffer("123")`

    Then: `sb.append(4)`

    is equivalent to

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

    Both yields `"1234"`



The Concatenation Operator (+)

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

- Concatenation by `StringBuffer`
  - `String x = "19" + 8 + 9;`

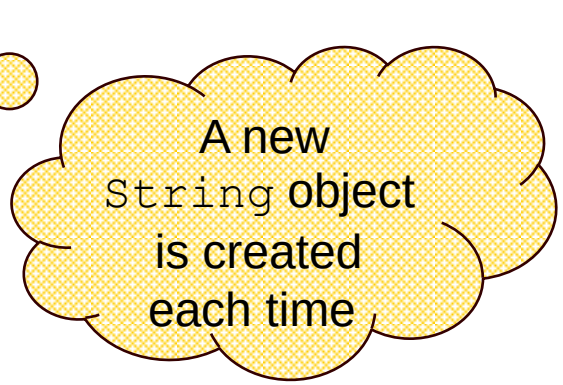
    is compiled to the equivalent of:

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

StringBuffer 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;
}
```

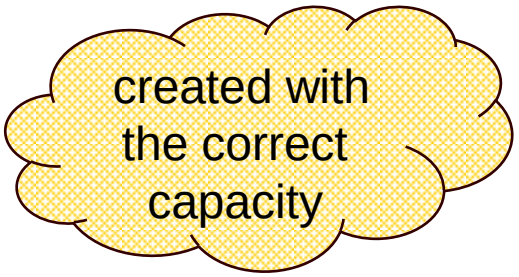


StringBuffer vs. String (cont.)

- More efficient version with StringBuffer:

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

StringBuffer vs. String (cont.)

- Even more efficient version:

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