תוכנה 1 בשפת Java

נושאים שונים בהורשה

רובי בוים ומתי שמרת

בית הספר למדעי המחשב
אוניברסיטת תל אביב

Today

- Static vs. Dynamic binding
- Equals / hashCode
- String Immutability (maybe)

Static versus run-time binding

- ```
  public class Account {
     public String getName(){...};
     public void deposit(int amount) {...};
  }

  public class SavingsAccount extends Account {
     public void deposit(int amount) {...};
  }
  ```
- ```
  Account obj = new Account();
  obj.getName();
  obj.deposit(…);

  obj = new SavingsAccount();
  obj.getName();
  obj.deposit(…);
  ```

Static binding (or early binding)

- Static binding: bind at compilation time
- Performed if the compiler can resolve the binding at compile time
  - Static functions
  - Access to member variables
  - Private methods
  - Final methods

Static binding example

```
public class A {
   public String someString = "member of A";
}
public class B extends A {
   public String someString = "member of B";
}

A a = new A();
A b = new B();
B c = new B();
System.out.println(a.someString);
System.out.println(b.someString);
System.out.println(c.someString);

Output:
   member of A
   member of A
   member of B
```

When to bind?

```
void func (Account obj) {
    obj.deposit();
}
```

- What should the compiler do here?
  - The compiler doesn't know which concrete object type is referenced by `obj`
  - the method to call can only be known at run time (*because of polymorphism*)
  - Run-time binding

תזכורת: המחלקה Object

```
package java.lang;

public class Object {
    public final native Class<?> getClass();

    public native int hashCode();

    public boolean equals(Object obj) {
        return (this == obj);
    }

    protected native Object clone() throws CloneNotSupportedException;

    public String toString() {
        return getClass().getName() + "@" +
               Integer.toHexString(hashCode());
    }
    ...
}
```

מה יודפס?

```
public class Name {
    private String first, last;
    ...

    public static void main(String[] args) {
        Name name1 = new Name("Mickey", "Mouse");
        Name name2 = new Name("Mickey", "Mouse");
        System.out.println(name1.equals(name2));

        List<Name> names = new ArrayList<Name>();
        names.add(name1);
        System.out.println(names.contains(name2));
    }
}
```

הבעיה

- רצינו השוואה לפי תוכן אבל לא דרסנו את equals
  - מימוש ברירת המחדל הוא השוואה של מצביעים

```
public class Object {
    ...
    public boolean equals(Object obj) {
        return (this == obj);
    }
    ...
}
```

החוזה של equals

- **רפלקסיבי**
  - `x.equals(x)` יחזיר `true`
- **סימטרי**
  - `x.equals(y)` יחזיר `true` אמ"מ `y.equals(x)` יחזיר `true`
- **טרנזיטיבי**
  - אם `x.equals(y)` מחזיר `true` וגם `y.equals(z)` מחזיר `true` אז `x.equals(z)`
- **עקבי**
  - סדרת קריאות ל `x.equals(y)` תחזיר true (או false) באופן עקבי אם מידע שדרוש לצורך ההשוואה לא השתנה
- **השוואה ל null**
  - `x.equals(null)` תמיד תחזיר `false`

מתכון ל equals

```
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Name other = (Name) obj;
    return first.equals(other.first) &&
           last.equals(other.last);
}
```

1. ודאו כי הארגומנט אינו מצביע לאובייקט הנוכחי
2. ודאו כי הארגומנט אינו null
3. ודאו כי הארגומנט הוא מהטיפוס המתאים להשוואה
4. המירו את הארגומנט לטיפוס הנכון
5. לכל שדה "משמעותי", בידקו ששדה זה בארגומנט תואם לשדה באובייקט הנוכחי

טעות נפוצה

- להגדיר את הפונקציה equals כך:

```
public boolean equals(Name name) {
    return first.equals(other.first) &&
           last.equals(other.last);
}
```

- זו אינה דריסה (overriding) אלא העמסה (overloading)
- שימוש ב @Override יפתור את הבעיה

אז הכל בסדר?

```
public class Name {
    ...
    @Override public equals(Object obj) {
        ...
    }

    public static void main(String[] args) {
        Name name1 = new Name("Mickey", "Mouse");
        Name name2 = new Name("Mickey", "Mouse");
        System.out.println(name1.equals(name2));

        List<Name> names = new ArrayList<Name>();
        names.add(name1);
        System.out.println(names.contains(name2));
    }
}
```

יודפס true

יודפס true

כמעט

```
public class Name {
    ...
    @Override public equals(Object obj) {
        ...
    }

    public static void main(String[] args) {
        Name name1 = new Name("Mickey", "Mouse");
        Name name2 = new Name("Mickey", "Mouse");
        System.out.println(name1.equals(name2));

        Set<Name> names = new HashSet<Name>();
        names.add(name1);
        System.out.println(names.contains(name2));
    }
}
```

יודפס true

יודפס false

hashCode ו equals

חובה לדרוס את hashCode בכל מחלקה שדורסת את equals!

החוזה של hashCode

- **עקביות**
  - מחזירה אותו ערך עבור כל הקריאות באותה ריצה, אלא אם השתנה מידע שבשימוש בהשוואת **equals** של המחלקה
- **שוויון**
  - אם שני אובייקטים שווים לפי הגדרת equals אזי hashCode תחזיר ערך זהה עבורם
- **חוסר שוויון**
  - אם שני אובייקטים אינם שווים לפי equals לא מובטח ש hashCode תחזיר ערכים שונים
  - החזרת ערכים שונים יכולה לשפר ביצועים של מבני נתונים המבוססים על hashing (לדוגמא, HashSet ו HashMap)

מימוש hashCode

```
@Override public int hashCode() {
    return 31 * first.hashCode() + last.hashCode();
}
```

- השתדלו לייצר hash כך שלאובייקטים שונים יהיה ערך hash שונה
- המימוש החוקי הגרוע ביותר (לעולם לא לממש כך!)

```
@Override public int hashCode() {
    return 42;
}
```

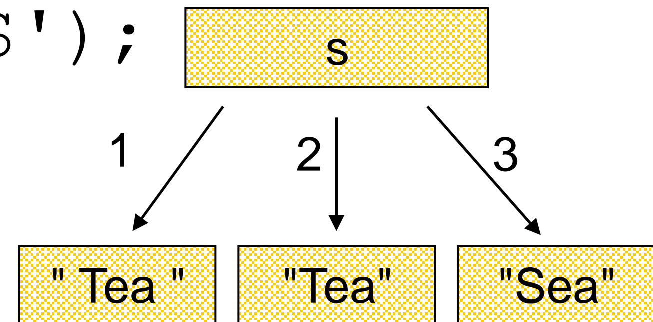
תמיכה באקליפס

- אקליפס תומך ביצירה אוטומטית (ומשולבת) של equals ו hashCode
  - בתפריט Source ניתן למצוא Generate hashCode() and equals()

String Immutability

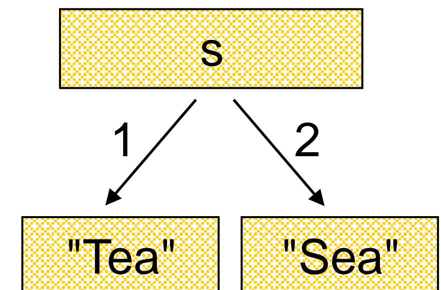
- Strings are constants

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



- A string reference may be set:

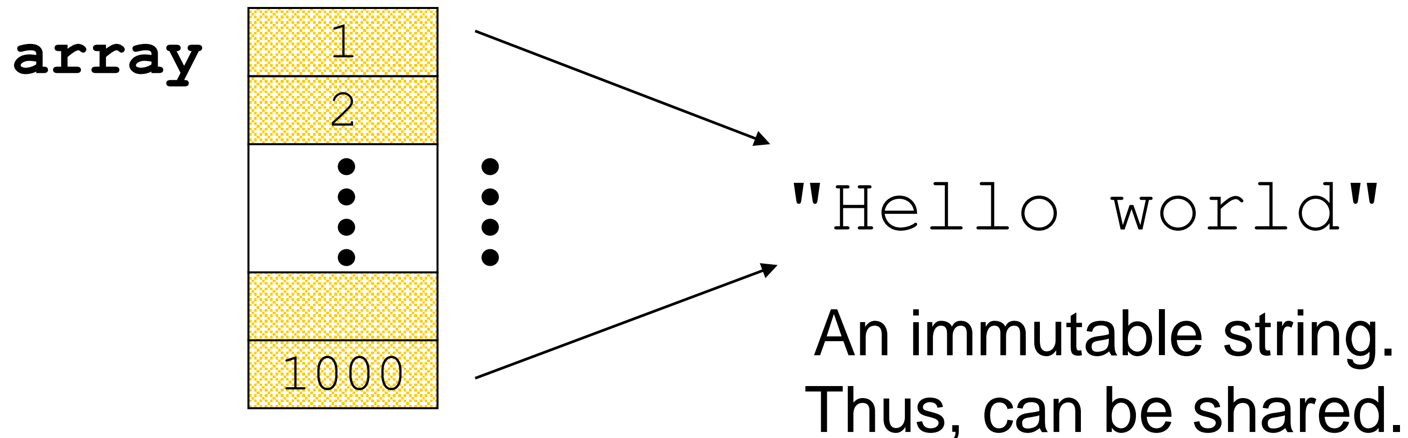
```
String s = "Tea";
s = "Sea";
```



String Interning

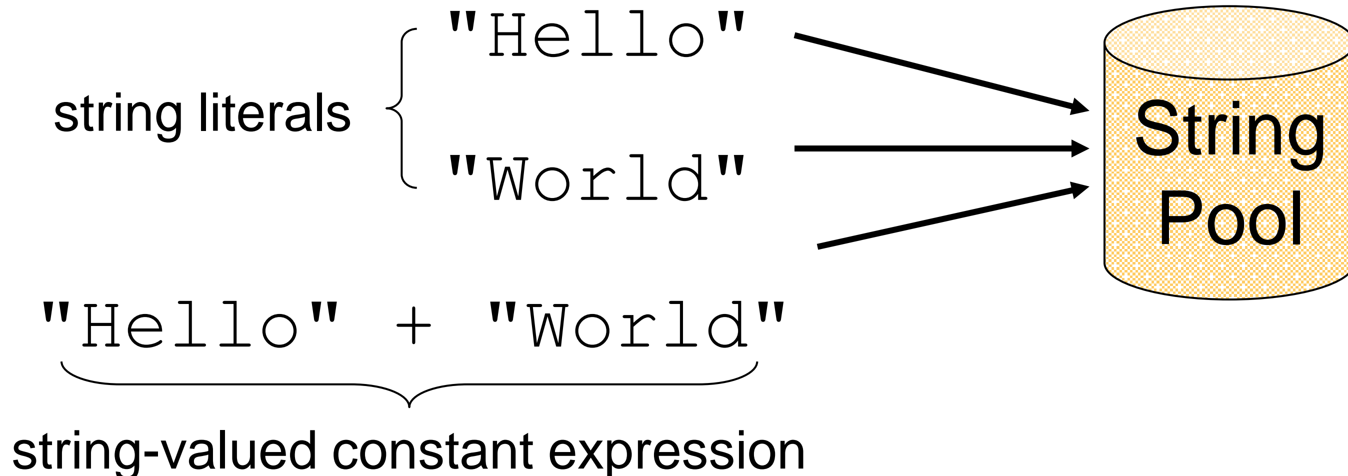
- Avoids duplicate strings

```
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 StringBuilder Class

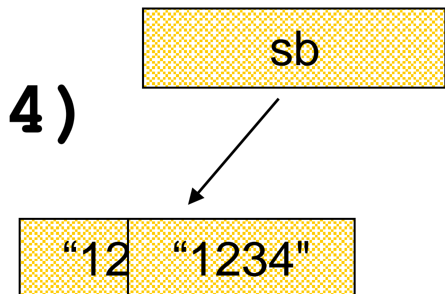
- Represents a **mutable** character string
- Main methods: **`append()`** & **`insert()`**
  - accept data of any type
  - If: **`sb = new StringBuilder("123")`**

    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`
  - `String x = "19" + 8 + 9;`

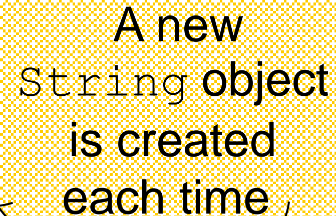
    is compiled to the equivalent of:

    ```
    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.)

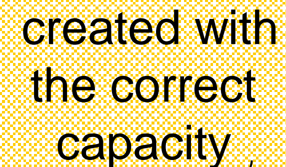
- More efficient version with StringBuilder:

```
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();
}
```



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.