Today

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

תוכנה 1 בשפת Java

תרגול מספר 9: נושאים שונים בהורשה

אסף זריצקי ומתי שמרת

בית הספר למדעי המחשב
אוניברסיטת תל אביב

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 versus run-time binding

```
public class Account {
   public static double interest = 0.01;
   public String getName(){...};
   public void deposit(int amount) {...};
 }

 public class SavingsAccount extends Account {
   public static double interest = 0.03;
   public void deposit(int amount) {...};
 }

Account obj = new Account();
obj.getName();
obj.deposit(…);
System.out.println(obj.accoutFrame);
obj = new SavingsAccount();
obj.getName();
obj.deposit(…);
System.out.println(obj.accoutFrame);
```


When to bind?

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

- What should the compiler do here?
  - `obj` is a pointer to different concrete `xxxAccount` object
  - the method to call can only be known at run time (*because of polymorphism*)
  - Run-time binding

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
```

Possible implementation of run-time binding (polymorphism)

- Not necessarily the exact Java implementation
- Each class has a `dvec` (***dispatch vector***)
  - `dvec` contains addresses of the class methods (that can be overriden)
- Every object has a pointer to it's class

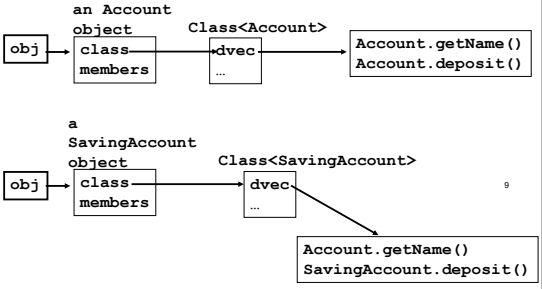
Run-time binding (or late binding)

- Binding
  - The translation of name into memory address
- Run-time binding
  - The translation is done at run-time
  - also known as
    - late binding
    - dynamic binding
    - virtual invocation
- Polymorphism depends on run-time binding

Dynamic binding – under the hood (simplified)

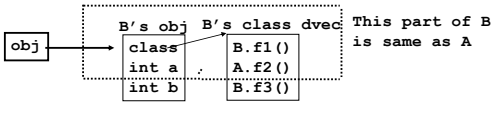
- Compile `obj.deposit()` to
  `obj.class.dvec[1](obj);`
- `obj` is a pointer to the object
- `obj.class` is a pointer to `obj's` runtime class (getClass())
- `obj.class.dvec` is a pointer to `dispatch vector`
- `obj.class.dvec[0]` is the 2nd slot in the `dvec`
- `deposit()` is the second method
- `obj.class.dvec[0](obj)` passes `obj` as '`this`' pointer
- If `obj` is an `Account`, then `Account.deposit()` is called
- If `obj` is a `SavingAccount`, then `SavingAccount.deposit()` is called

Possible implementation of run-time binding (polymorphism)



Why B can be treated as A?

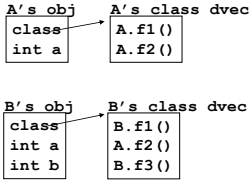
- Remember the "*is a*" relation?
- The top part of `B` is same as `A`, so it can be treated as `A` (upcasting, and hence polymorphism)



Another example

```
class A {
   public final void f0(){…};
   public void f1(){…};
   public void f2(){…};
   private int a;
}

class B extends A {
   public void f1();
   public void f3();
   protected int b;
}
```



f0 is a method that can not be inherited
f1() is overridden by B
f2() has not been overridden
f3() is a new method in B

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 {
    ...
    @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));
    }
}
```


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;
}
```

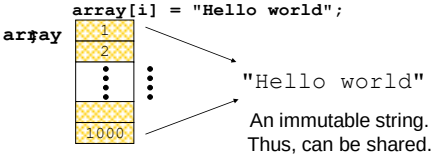

תמיכה באקליפס

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

String Interning

- Avoids duplicate strings

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



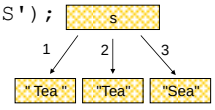
An immutable string.
Thus, can be shared.

26

String Immutability

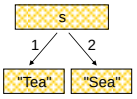
- Strings are constants

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



- A string reference may be set:

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



String Constructors

- Use implicit constructor:

  `String s = "Hello";`

  (string literals are interned)

  Instead of:

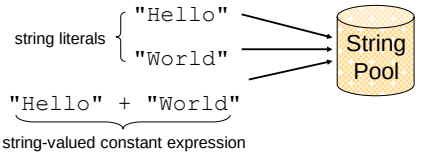
  `String s = new String("Hello");`

  (causes extra memory allocation)

28

String Interning (cont.)

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



27

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();
```

30

The StringBuffer Class

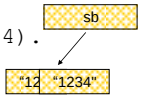
- Represents a **mutable** character string
- Main methods: `append()` & `insert()`
  - accept data of any type
  - If: `sb = new StringBuffer("123")`

    Then: `sb.append(4)`

    is equivalent to

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

    Both yields `"1234"`



29

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();
}
```

no new Objects

32

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;
}
```

A new `String` object is created each time

31

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();
}
```

created with the correct capacity

33