Today

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

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

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

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 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
The content of the page includes a code snippet for the `Object` class in Java, demonstrating the `equals` method and its contract. The definitions of the `equals` method are translated into Hebrew, discussing the method's requirements and contract.

The code snippet is as follows:

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

The Hebrew text explains the `equals` method's contract:

1. Reflexive: `x.equals(x)` returns true
2. Symmetric: `x.equals(y)` returns true if and only if `y.equals(x)` returns true
3. Transitive: If `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` returns true
4. Consistency: A sequence of calls to `x.equals(y)` returns true (or false) consistently if information used for comparison does not change
5. Null: `x.equals(null)` always returns false

The Hebrew text also highlights common mistakes:

- Misdefining `equals` method as overridden instead of overloaded.
- Using `@Override` annotation to resolve the issue.

The page further discusses the differences between `equals` and `==` operators and provides tips on testing for equality.

The page concludes with a section on testing equality: `sequality t:atx f: equalsOrNot() { ...
... return true; }`

It also mentions that the `equals` method is designed to provide a consistent way to compare objects, especially when the objects are complex or have different types.
כמובן

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 true 

 azimuth}  

hashCode של equals

dירוגיatz: 
  מילה אחת עבור כל תקריב.qa כולל 
  המילה של הקודיעה risky. 

hashCode וequals

הצבה של hashCode בכל מקלה! equals צידורsten את

הザー בעבר

hashCode באת

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

@override public int hashCode() { 
    return 42; 
} 

_hashes ובsetData

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

@override public int hashCode() { 
    return 42; 
} 

почום והזזות ייצרו שולחוני שים יונישה

hashCode המיים

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

@override public int hashCode() { 
    return 42; 
}
String Interning

- Avoids duplicate strings

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

array

"Hello world"

An immutable string. Thus, can be shared.

String Interning (cont.)

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

```
String literals

"Hello"
"World"

"Hello" + "World"

string-valued constant expression
```

String Constructors

- Use implicit constructor:

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

(string literals are interned)

Instead of:

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

(cause extra memory allocation)

String Immutability

- Strings are constants

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

A string reference may be set:

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

String Constructors (cont.)

- Use implicit constructor:

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

(string literals are interned)

Instead of:

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

(cause extra memory allocation)

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:

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

The String Builder 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"
StringBuilder vs. String (cont.)

- More efficient version with StringBuilder:

```java
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. 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.

StringBuilder vs. String (cont.)

- Even more efficient version:

```java
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();
}
```