

תכנות מתקדם בשפת Java

עיצוב מחלקה

(וקצת עיצוב על פי חוזה)

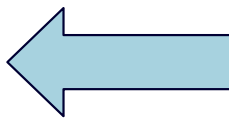
אורנית דרור ואוהד ברזילי

תכנון תוכנה למערכת בנקאית

תכנון מערכת תוכנה עוסק במיפוי בין עולם הבעיה ועולם הפתרון

□ עולם הפתרון:

- שפת תכנות
- עצמים
- מחלקות
- מתודות
- שדות



□ עולם הבעיה:

- בנקים
- לקוחות
- משיכות, הפקדות
- חשבונות
- יתרות

מחלקה לייצוג חשבון בנק

- בגישה מוכוונת עצמים כל שם עצם מעולם הבעיה הוא מועמד לייצוג ע"י מחלקה
- נתכנן מחלקה, `BankAccount`, לייצוג חשבון בנק
- ננסה להפוך את התאור המילולי והתפיסה האינטואיטיבית שלנו של חשבון בנק לרכיב תוכנה
- תאור הפעולות יתבטא בחוזה ובמתודות המחלקה
- יש להזהר לא להצמד יותר מדי לתאור העולם האמיתי (דוגמא: פקיד בנק שעושה הכל)

המצב הפנימי

- המצב הפנימי של עצם מיוצג ע"י נתוניו (שדותיו)
- שדות עצם הם בד"כ עם הרשאת גישה פרטית
- במקרה של חשבון בנק – היתרה
- מאיזה טיפוס?

```
public class BankAccount {  
    ...  
    private ??? balance;  
}
```

המצב הפנימי

- המצב הפנימי של עצם מיוצג ע"י נתוניו (שדותיו)
- שדות עצם הם בד"כ עם הרשאת גישה פרטית
- במקרה של חשבון בנק – היתרה
- מאיזה טיפוס?

```
public class BankAccount {  
    ...  
    private double balance;  
}
```

שרותי מחלקה

- ישנם 3 סוגים של מתודות (שרותים, פונקציות):
 - פקודות (commands, transformers, mutators)
 - מבצעות שינוי במצב המופשט (abstract state) של העצם
 - כגון: משיכה, הפקדה
 - שאילתות (queries, accessors)
 - מחזירות ערך ללא שינוי המצב המופשט
 - כגון: בירור יתרה
 - בנאים (constructors)
 - יצירת עצם חדש
 - כגון: יצירת חשבון חדש

חתימה של פקודות

- בד"כ פקודות אינן מחזירות ערך (גם לא ערך שגיאה) וחתימתן היא עם טיפוס ערך מוחזר `void`
- לפעמים פקודות מחזירות הפנייה לעצם הנוכחי (`this`)
 - בעד: מאפשר הרכבה של פקודות
 - נגד: מטשטש את ההבחנה בין שאילתה ופקודה

```
x.command1();  
x.command2();  
x.command3();
```

} `x.command1().command2().command3();`

e.g. `new StringBuilder().append("19").append(84).toString();`

שאלות BankAccount

□ ברור יתרה:

■ ארגומנטים?

■ מה טיפוס הערך המוחזר?

■ תנאי קדם? תנאי בתר?

□ פרטים על החשבון:

■ מספר חשבון?

■ פרטים על בעל החשבון?

□ תעודת זהות?

□ גיל?

שאלות BankAccount

```
public class BankAccount {  
    public double getBalance() {  
        return balance;  
    }  
}
```

□ בעולם ה"אקדמי" מקובל לגשת לנתון field בעזרת המתודה field()

```
public long getAccountNumber() {  
    return accountNumber;  
}
```

□ בשפת Java השתרשה המוסכמה כי הגישה לשדה field תעשה בעזרת המתודה getField()

```
public Customer getOwner () {  
    return owner;  
}
```

□ שמירה על מוסכמה זו הכרחית בסביבות GUI Builders ו- JavaBeans

```
private double balance;  
private long accountNumber;  
private Customer owner;  
}
```

setter/getter

- לא כל שדה עם נראות פרטית (**private**) צריך setter/getter ציבורי

- יצירה 'אוטומטית' של שרותים אלו עבור כל שדה פוגמת בעקרון הסתרת המידע
 - ועם זאת, עדיין יש חשיבות לגישה לנתונים דרך מתודות. מדוע?

- למשל: נתבונן בשדה: `private double balance`
 - האם דרוש `getter`? כן, זהו חלק מהממשק של חשבון בנק
 - האם דרוש `setter`? לא בהכרח, פעולות של משיכה או הפקדה אמנם משפיעות על היתרה, אבל פעולה של שינוי יתרה במנותק מהן אינה חלק מהממשק

פקודת ה-'להפקיד'

□ המתודה: deposit

□ סכום הכסף המופקד מתווסף ליתרה בחשבון

■ ארגומנטים?

■ ערך מוחזר?

■ תנאי קדם?

■ תנאי בתר?

מוסכמה: שמות פקודות
הם שמות פועל

פקודת ה-'להפקיד'

```
/**
 * Makes a deposit to the current account
 * @pre amount > 0 , "amount is positive"
 * @post getBalance() == $prev(getBalance()) + amount ,
 *       "balance updated according to deposit"
 */
public void deposit(double amount) {
    balance += amount;
}
```

פקודת ה-'למשוך'

- המתודה: withdraw
- סכום הכסף המבוקש יורד מיתרת החשבון. אין באפשרותו של הלקוח להיכנס למצב של משיכת יתר
 - ארגומנטים?
 - ערך מוחזר?
 - תנאי קדם?
 - תנאי בתר?
- תנאיי קדם לא יבדקו בגוף המתודה – זהו תכנות מתגונן והוא שגוי בכמה היבטים

פקודת ה-'למשוך'

```
/**
 * Withdraw amount from the current account
 * @pre amount <= getBalance() , "can't overdraft"
 * @pre 0 < amount , "amount is positive"
 * @post getBalance() == $prev(getBalance()) - amount ,
 *       "balance updated according to withdraw"
 */
public void withdraw(double amount) {
    balance -= amount;
}
```

דיון – העברה בנקאית

- נדון במספר חלופות למימוש העברת סכום מחשבון לחשבון
- אפשרות א': העמסת withdraw ו-deposit שיקבלו 2 ארגומנטים: סכום והפנייה לחשבון נוסף. לדוגמא:

```
/**
 * Makes a transfer of amount from other to the current account
 * @pre 0 < amount , "amount is positive"
 * @pre amount <= other.getBalance(), "other can't overdraft"
 * @post getBalance() == $prev(getBalance()) + amount,
 *       "balance updated"
 * @post other.getBalance() == $prev(other.getBalance()) -
 *       amount,
 *       "balance of other updated"
 */
public void deposit(double amount, BankAccount other) {
    other.withdraw(amount);
    balance += amount;
}
```

דיון – העברה בנקאית

ניתן לתת למתודות שמות מפורשים יותר, כגון `transferTo` ■
: `transferFrom` א

```
/**  
 * Makes a transfer of amount from current to  
 * the other account...  
 */  
public void transferTo(double amount, BankAccount other) {  
    other.deposit(amount);  
    balance -= amount;  
}
```


דיון – העברה בנקאית

□ אפשרות ב' – מתודה סטטית (הסבר בהמשך הקורס) שתקבל שני חשבונות בנק ותבצע ביניהם העברה:

```
/**
 * Makes a transfer of amount from one account to the other
 * @pre 0 < amount <= from.getBalance() , "from can't overdraft"
 * @post to.getBalance() == $prev(to.getBalance()) + amount
 * @post from.getBalance() == $prev(from.getBalance()) - amount
 */
public static void transfer(double amount, BankAccount from,
                             BankAccount to) {
    from.withdraw(amount);
    to.deposit(amount);
}
```

שמורת המחלקה (class invariant)

- צריכה להתקיים "תמיד"
 - לפני ואחרי ביצוע כל מתודה ציבורית
 - אחרי הבנאי
- במחלקה חשבון בנק:
 - חשבון חייב להיות עם יתרה אי שלילית
 - לכל חשבון קיים מספר מזהה במערכת
 - לכל חשבון יש בעלים

שמורת BankAccount

```
/**
 * This class represents a bank account
 * @inv getBalance() >= 0,
 *         "can't overdraft"
 * @inv getAccountNumber() > 0 ,
 *         "an account must have an identifier"
 * @inv getOwner() != null ,
 *         "an account must have owner"
 */
public class BankAccount {
    ...
}
```

בנאי

- תפקיד הבנאי הוא ליצור עצם חדש ולהביא אותו למצב המקיים את שמורת המחלקה
- בנאי לא אמור לכלול לוגיקה נוספת פרט לכך
- במחלקה `BankAccount`: בנאי ברירת המחדל יוצר עצם שאינו מקיים את השמורה!
- יש דברים שאינם באחריות המחלקה. למשל:
 - מי דואג שמספרי החשבון יהיו תקינים? (למשל שונים זה מזה)
 - מי מנהל את מאגר הלקוחות?
 - הכמסה (encapsulation)

בנאי BankAccount

```
/**
 * Constructs a new account and sets its owner and
 * identifier
 * @pre id > 0, "account number must be positive"
 * @pre customer != null,
 *         "an account must have an owner"
 * @post getOwner() == customer,
 *         "argument was assigned"
 * @post getAccountNumber() == id,
 *         "argument was assigned"
 */
public BankAccount(Customer customer, long id) {
    accountNumber = id;
    owner = customer;
}
```

final

- מכיוון שחשבון מזוהה חד-חד ערכית עם עצם של `accountNumber` נהפוך שדה זה ל- `final` :

```
final private long accountNumber;
```

- את השדה (*blank final*) יש לאתחל פעם אחת בדיוק, בתוך הבנאי של המחלקה, כפי שאנו אכן עושים
- כעת, מרגע שנוצר עצם, שפת התכנות אוכפת את הצימוד בין העצם והמזהה שלו

חזזה מימוש

- תנאי הקדם מיועדים ללקוח ולכן אסור להם להכיל רכיבים שאינם זמינים לו (כגון מתודות או שדות private)
- תנאי הבתר ושמורת המחלקה עשויים להכיל טענות "לצורכי פנים" שיסומנו ב: `@imp_post`, `@imp_inv` לדוגמא:

```
/**
 * @imp_post $ret == balance ,
 *           "consistency of representation"
 */
public double getBalance() {...}
```

balance הוא שדה private ולכן אינו מיועד ללקוחות

שמורת מימוש של BankAccount

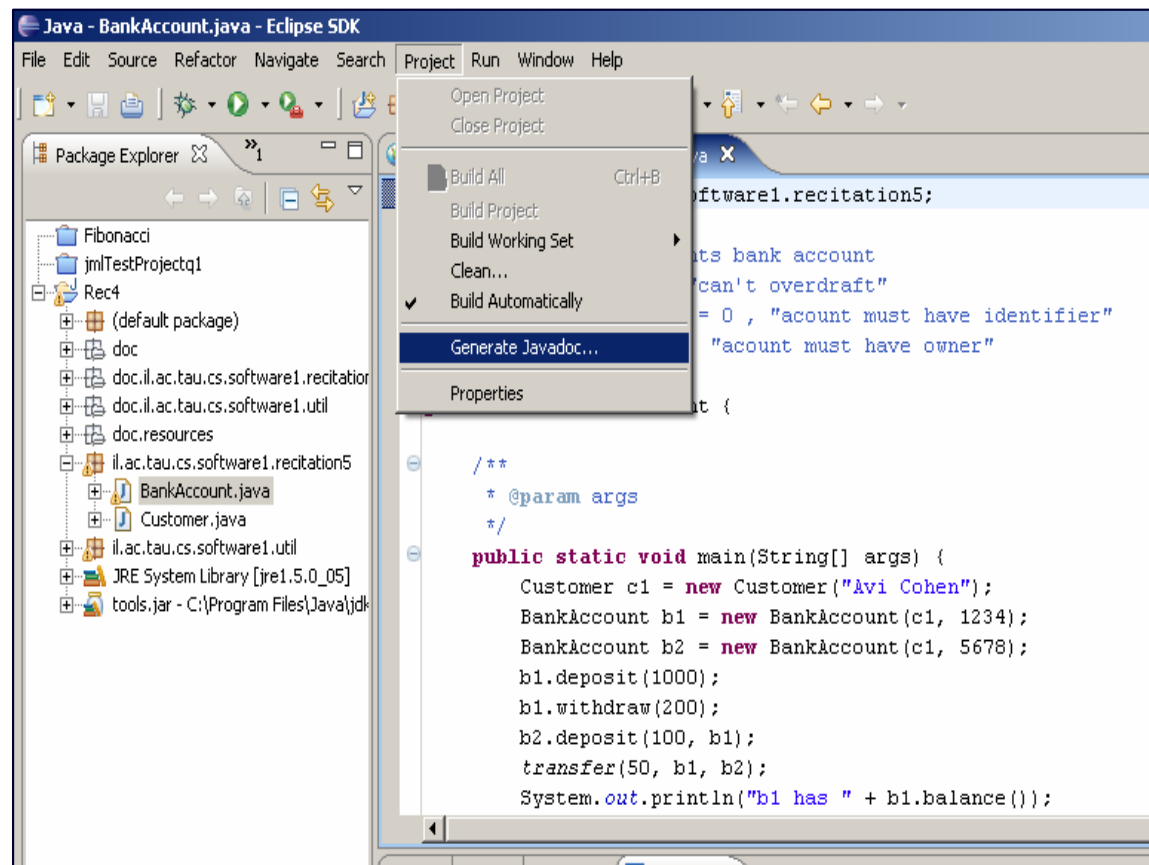
```
/**
 * This class represents a bank account
 * @imp_inv getBalance() == balance,
 *     "balance interface is consistent with representation"
 * @imp_inv getOwner() == owner
 *     "owner interface is consistent with representation"
 */
public class BankAccount {
    ...
}
```

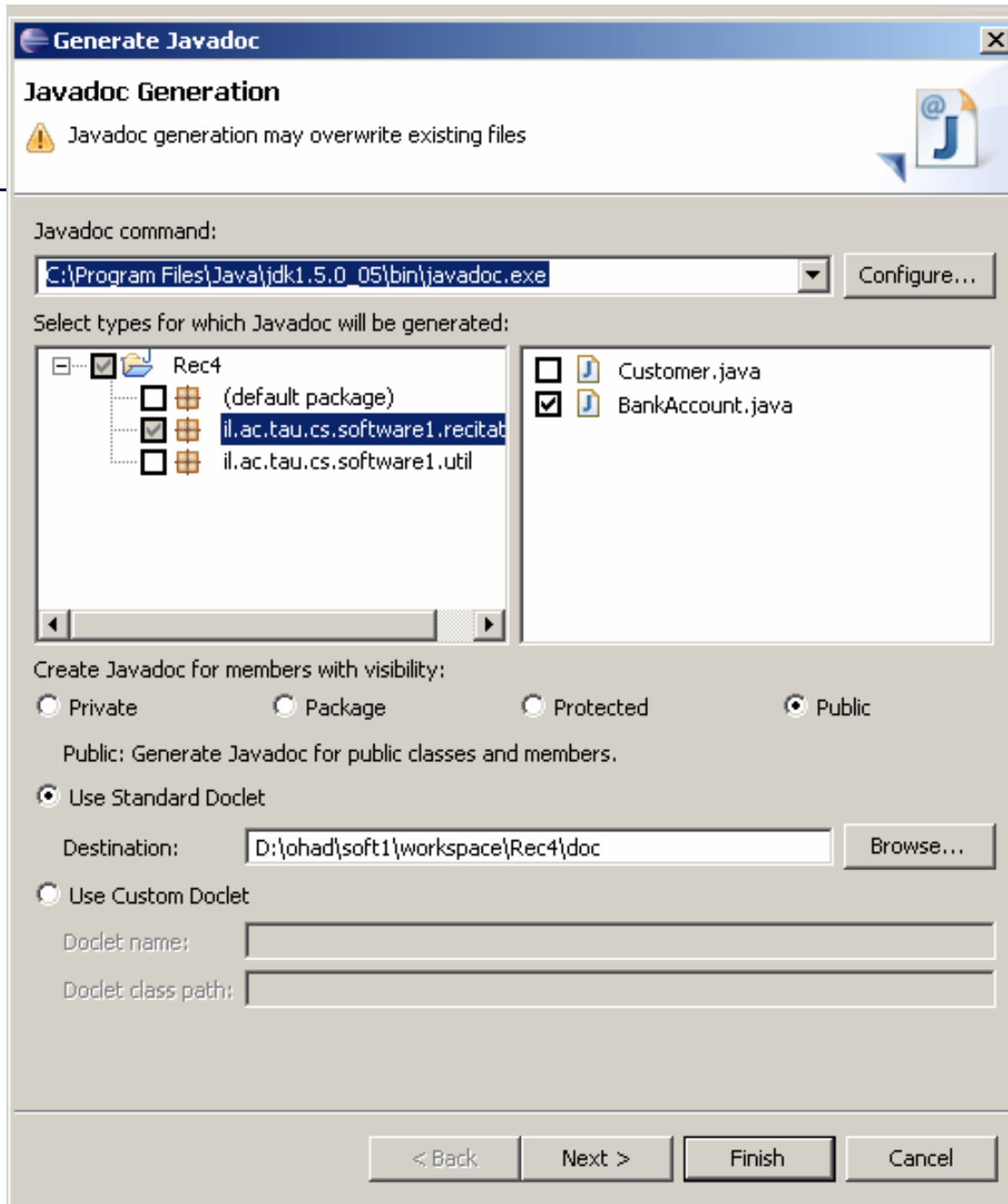

יצירת תיעוד אוטומטי

עבודה עם javadoc והוספת תגיות חוזה

javadoc

- כדי לחולל תיעוד אוטומטי עבור הקוד שכתבנו נבחר בסרגל הכלים: Project ->Generate Javadoc...





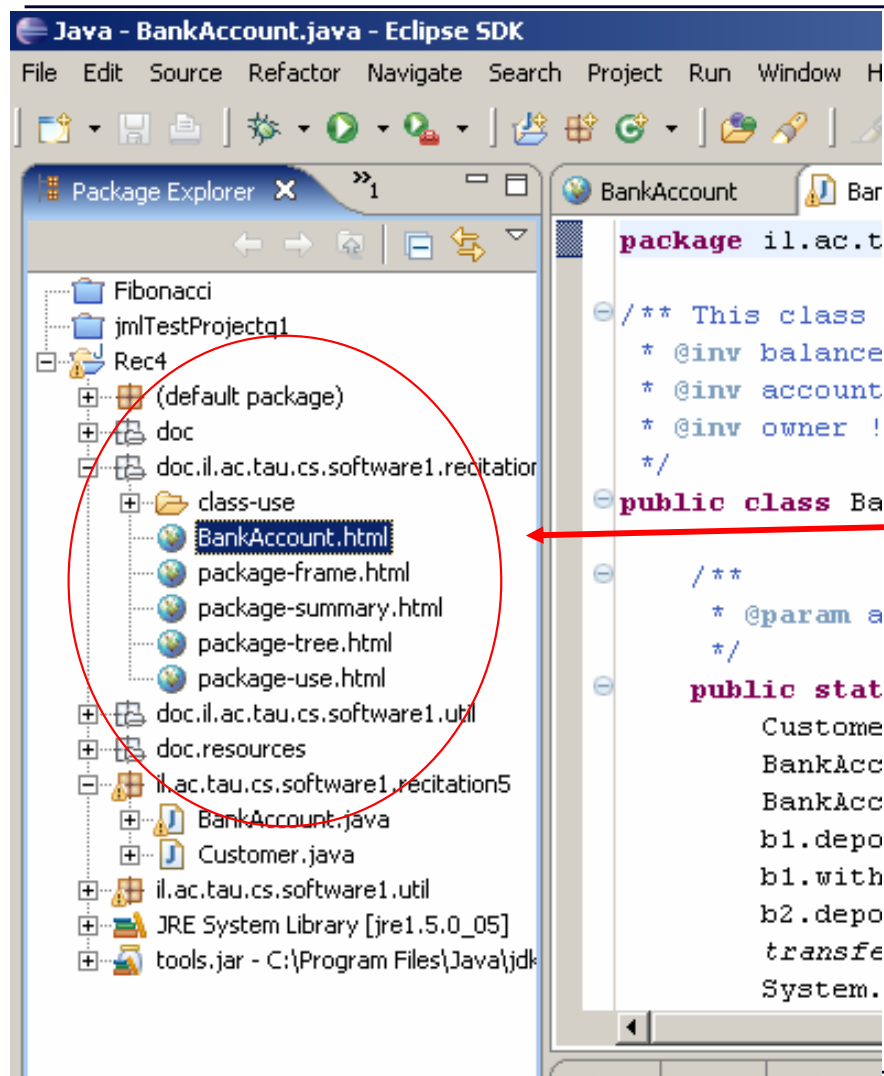
javadoc

- נזין (בעזרת הכפתור `configure` או בשורת המלל) את מיקומה של התוכנית `javadoc`. תוכנית זו כלולה בחבילת ה-`JDK` (`Java SDK`) שהורדנו מאתר חברת `Sun`
- מיקום טיפוס של החבילה הוא ב:
`C:\Program Files\Java\jdk1.5.x_xx\bin\javadoc.exe`
(ה-`x`ים מסמנים את מספר הגרסה)

javadoc

- ☑ סמנו את הקבצים שברצונכם לתעד ב-
- ☐ בחרו את רמת הגישה אשר ממנה אתם מעוניינים לתעד. לדוגמא:
- **public**: רק מתודות ושדות ציבוריים יופיעו בתיעוד. תיעוד זה מיועד ללקוחות של המחלקה. נקרא גם API
- **private**: כל המתודות והשדות יכללו בתיעוד. תיעוד זה מיועד למפתחי המחלקה
- לצורך הגשת התרגילים בקורס זה יש לבחור ב `private`
- ☐ האפשרות Use Standard Doclet תיצור תיעוד אוטומטי בפורמט HTML במיקום שיצויין בשורת ה Destination

javadoc



- לחיצה על Finish
- תיצור את התיעוד המבוקש
- תיקייה הכוללת את דפי התיעוד תופיע בסייר החבילות
- בסביבת העבודה
- לחיצה על דפי הhtml תפתח אותם בסביבת העבודה

הכללת החוזה בתיעוד

□ מחולל התיעוד התקני אינו 'מכיר' את התגיות:

`@pre`, `@post`, `@inv`, `@imp_inv`, `@imp_post`

□ ניתן להוסיף תגיות אלו ע"י רישום שלהן בתוכנה

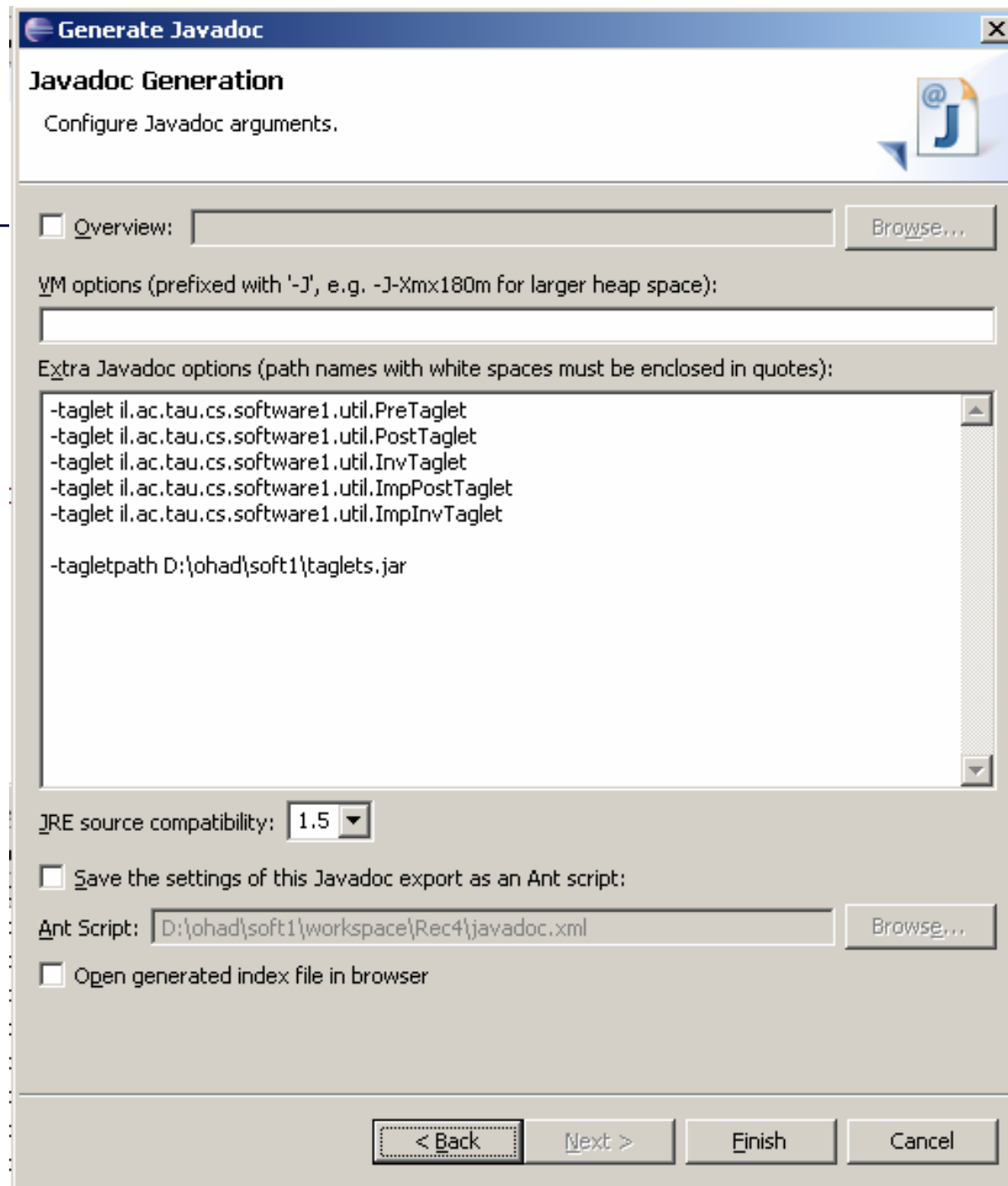
javadoc באופן הבא:

■ הורידו את הקובץ [taglets.jar](#) מאתר הקורס ומקמו

אותו בתיקייה לבחירתכם

□ למשל: `D:\ohad\soft1\taglets.jar`

□ בתפריט `Generate Javadoc...` לחצו פעמיים על `Next`



הכללת החוזה בתיעוד

□ הזינו בחלון Extra Javadoc options את הפרטים הבאים:

```
-taglet il.ac.tau.cs.software1.util.PreTaglet
-taglet il.ac.tau.cs.software1.util.PostTaglet
-taglet il.ac.tau.cs.software1.util.InvTaglet
-taglet il.ac.tau.cs.software1.util.ImpPostTaglet
-taglet il.ac.tau.cs.software1.util.ImpInvTaglet

-tagletpath D:\ohad\soft1\taglets.jar
```

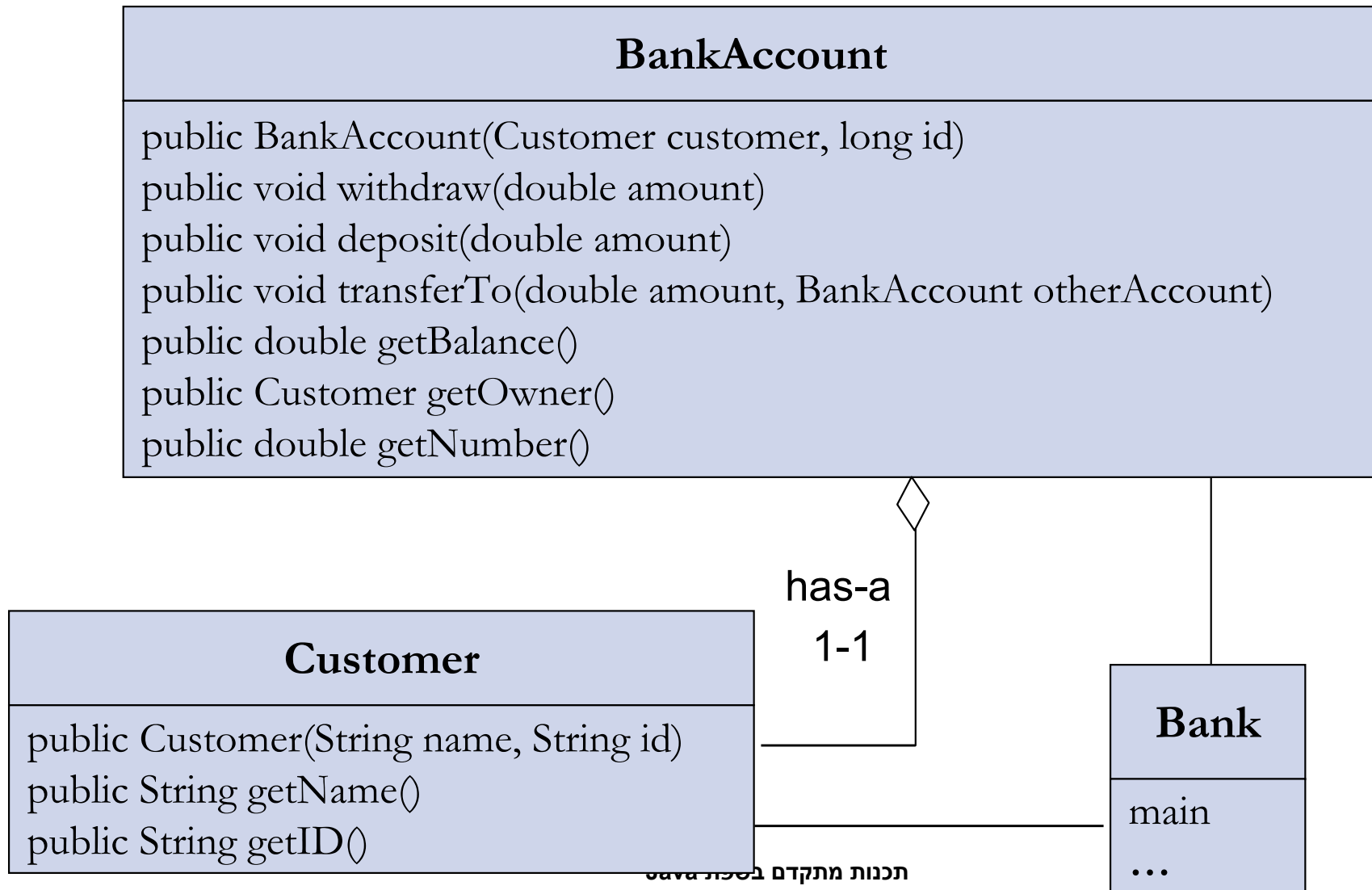
□ עדכנו את מיקום הקובץ ה `taglets.jar` לפי מיקומו
במחשב שלכם

□ לסיום ליחצו על **Finish**



Putting it all together

Class Diagram



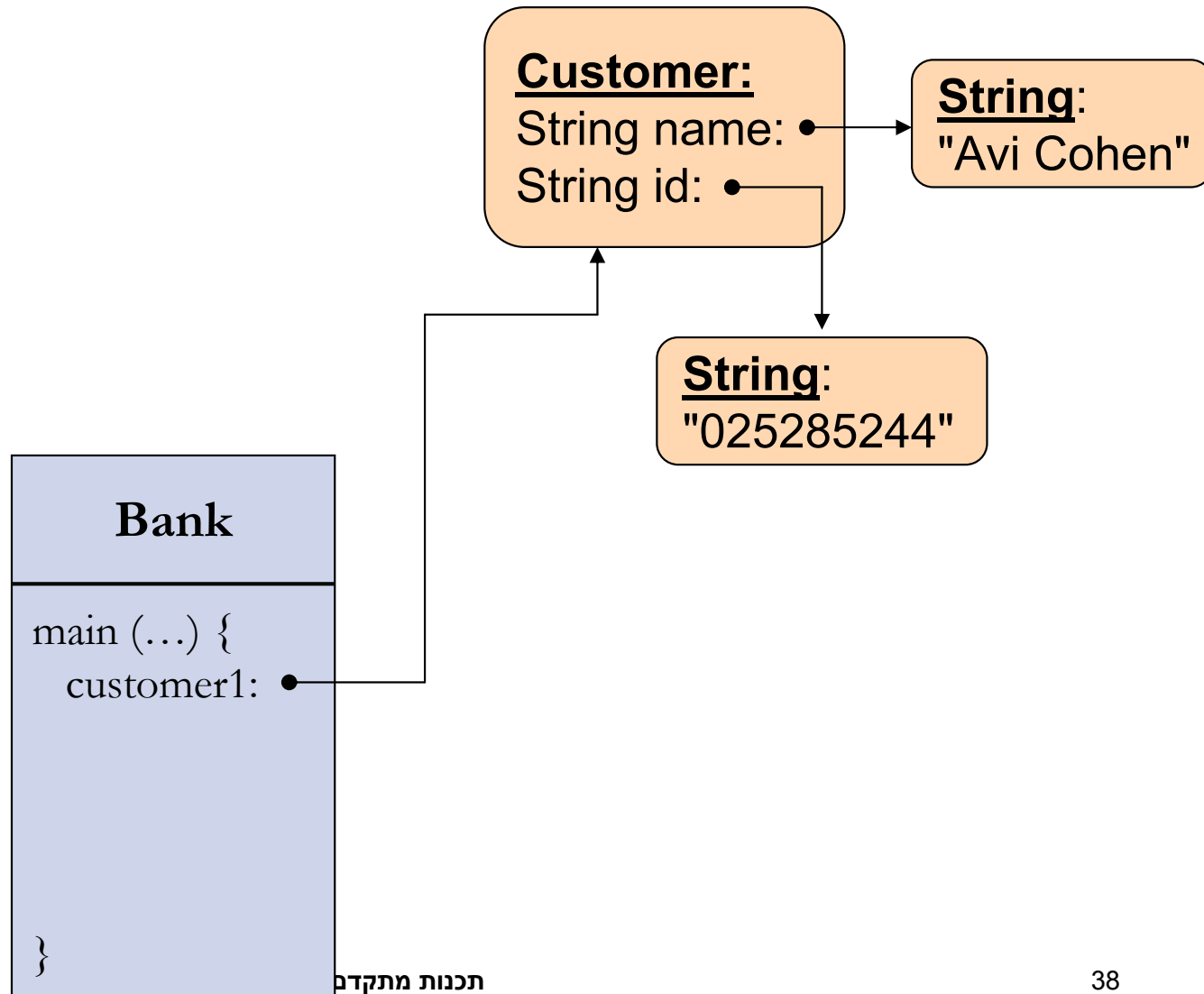
The Customer Class

```
public class Customer {  
    public Customer(String name, String id) {  
        this.name = name;  
        this.id = id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getID() {  
        return id;  
    }  
  
    private String name;  
    private String id;  
}
```

Toy Bank Program

```
public class Bank {  
    public static void main(String[] args) {  
        → Customer customer1 = new Customer("Avi Cohen", "025285244");  
        Customer customer2 = new Customer("Rita Stein", "024847638");  
  
        BankAccount account1 = new BankAccount(customer1, 1234);  
        BankAccount account2 = new BankAccount(customer2, 5678);  
        BankAccount account3 = new BankAccount(customer2, 2984);  
  
        account1.deposit(1000);  
        account2.deposit(500);  
        account1.transferTo(100, account3);  
        account2.withdraw(300);  
  
        System.out.println("account1 has " + account1.getBalance());  
        System.out.println("account2 has " + account2.getBalance());  
    }  
}
```

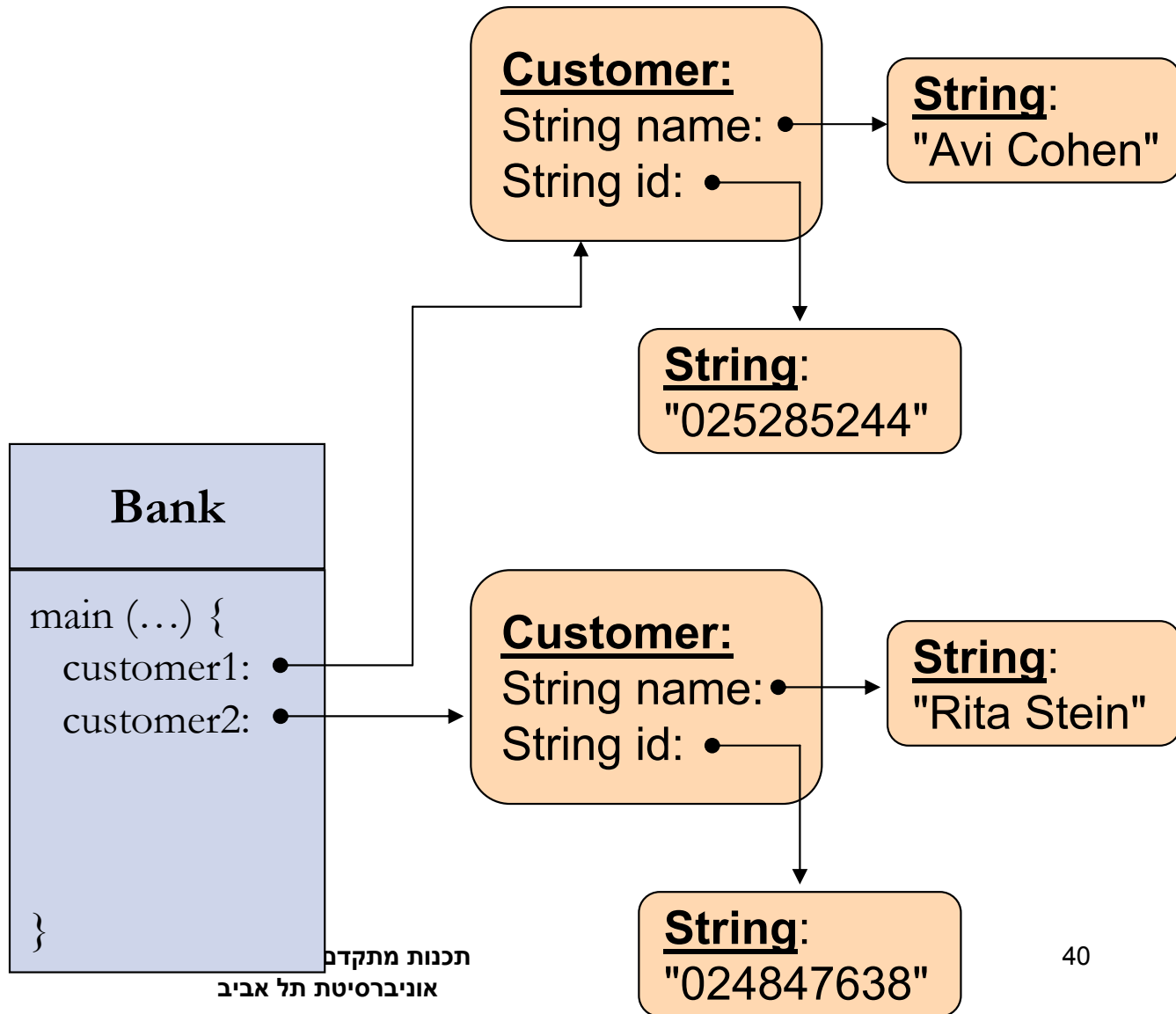
Object Diagram



Toy Bank Program

```
public class Bank {  
    public static void main(String[] args) {  
        Customer customer1 = new Customer("Avi Cohen", "025285244");  
        → Customer customer2 = new Customer("Rita Stein", "024847638");  
        BankAccount account1 = new BankAccount(customer1, 1234);  
        BankAccount account2 = new BankAccount(customer2, 5678);  
        BankAccount account3 = new BankAccount(customer2, 2984);  
  
        account1.deposit(1000);  
        account2.deposit(500);  
        account1.transferTo(100, account3);  
        account2.withdraw(300);  
  
        System.out.println("account1 has " + account1.getBalance());  
        System.out.println("account2 has " + account2.getBalance());  
    }  
}
```

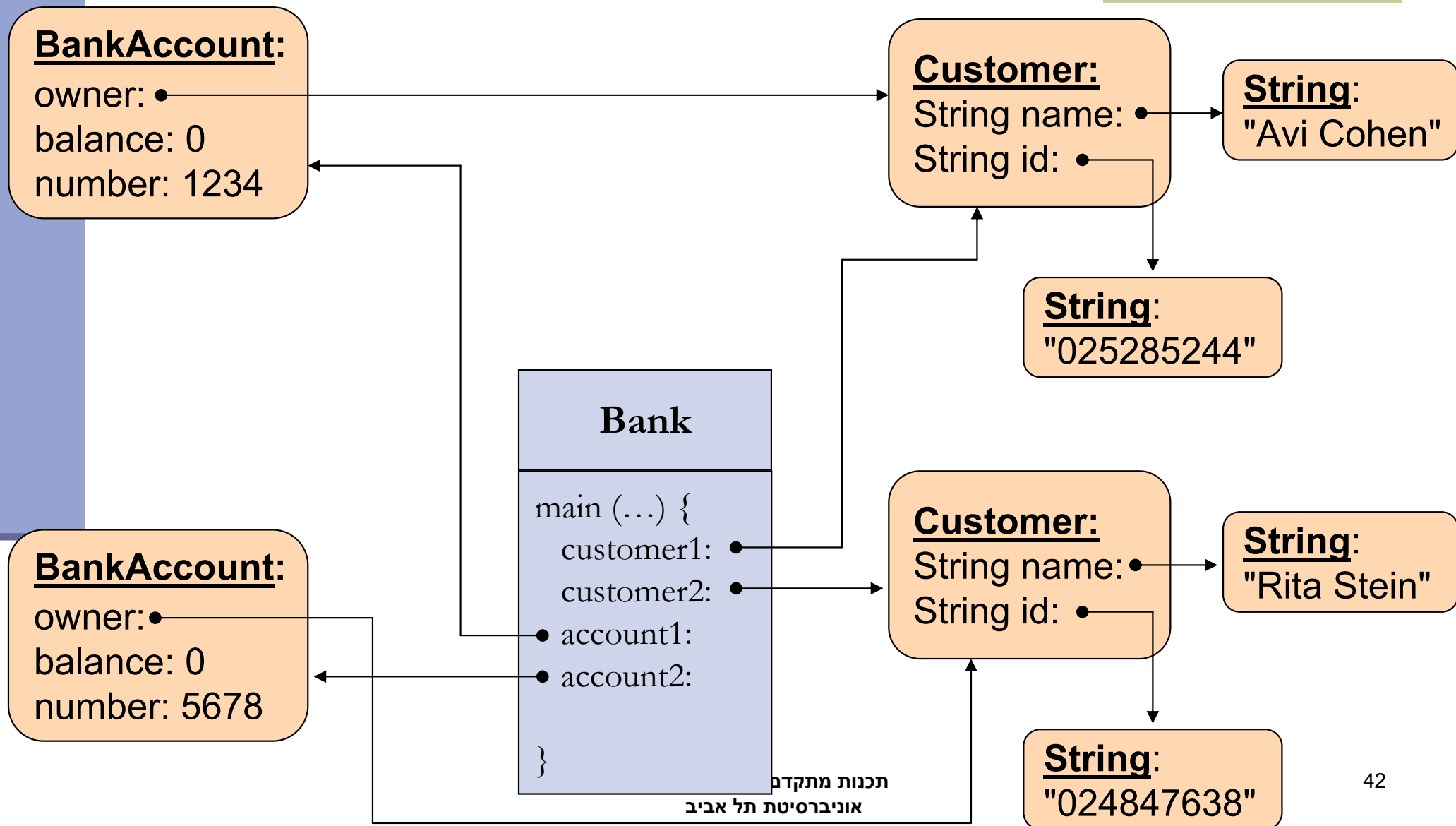
Object Diagram



Toy Bank Program

```
public class Bank {  
    public static void main(String[] args) {  
        Customer customer1 = new Customer("Avi Cohen", "025285244");  
        Customer customer2 = new Customer("Rita Stein", "024847638");  
        → BankAccount account1 = new BankAccount(customer1, 1234);  
        BankAccount account2 = new BankAccount(customer2, 5678);  
        BankAccount account3 = new BankAccount(customer2, 2984);  
  
        account1.deposit(1000);  
        account2.deposit(500);  
        account1.transferTo(100, account3);  
        account2.withdraw(300);  
  
        System.out.println("account1 has " + account1.getBalance());  
        System.out.println("account2 has " + account2.getBalance());  
    }  
}
```

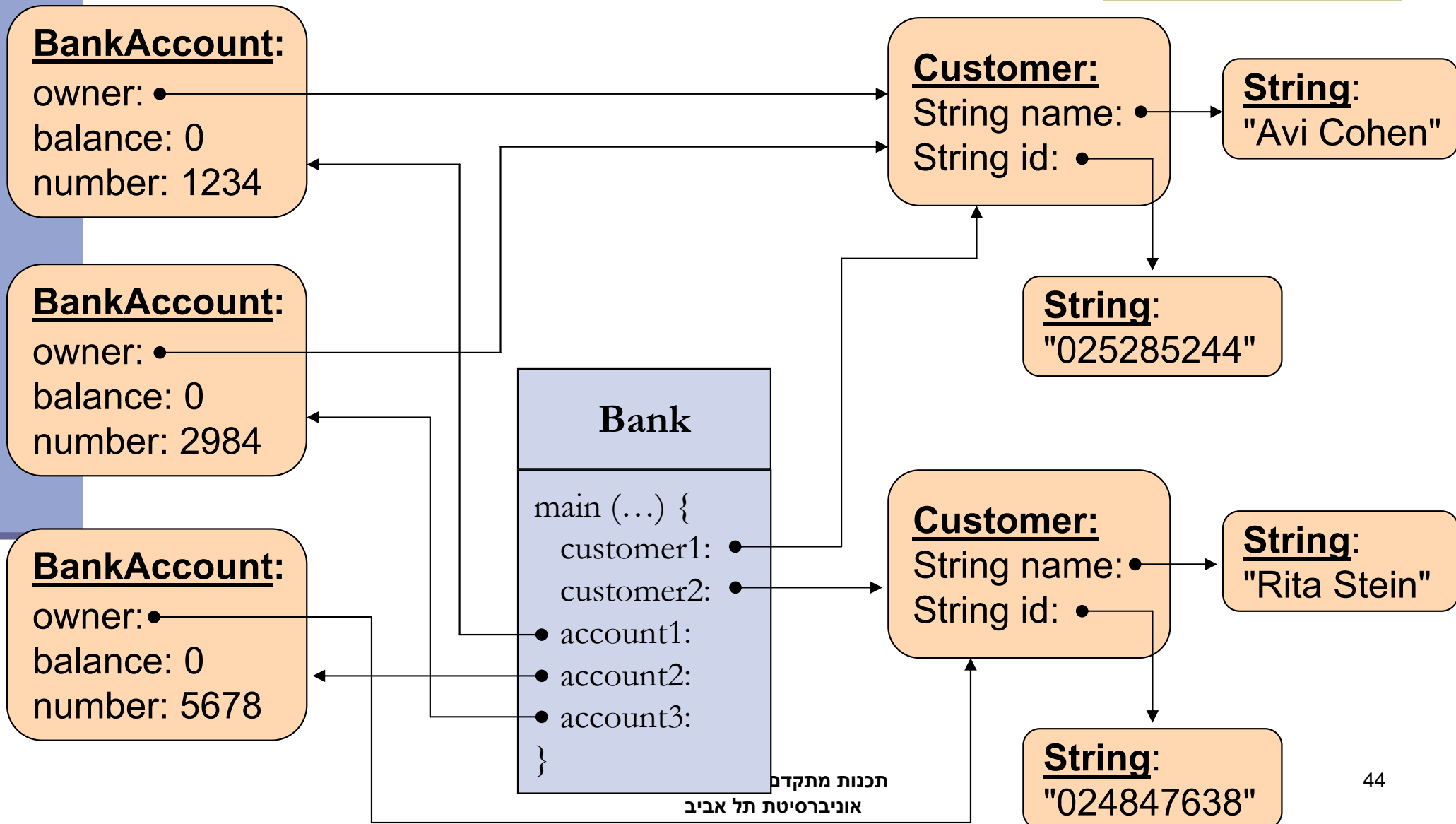
Object Diagram



Toy Bank Program

```
public class Bank {  
    public static void main(String[] args) {  
        Customer customer1 = new Customer("Avi Cohen", "025285244");  
        Customer customer2 = new Customer("Rita Stein", "024847638");  
        BankAccount account1 = new BankAccount(customer1, 1234);  
        BankAccount account2 = new BankAccount(customer2, 5678);  
        → BankAccount account3 = new BankAccount(customer1, 2984);  
        account1.deposit(1000);  
        account2.deposit(500);  
        account1.transferTo(100, account3);  
        account2.withdraw(300);  
        System.out.println("account1 has " + account1.getBalance());  
        System.out.println("account2 has " + account2.getBalance());  
    }  
}
```

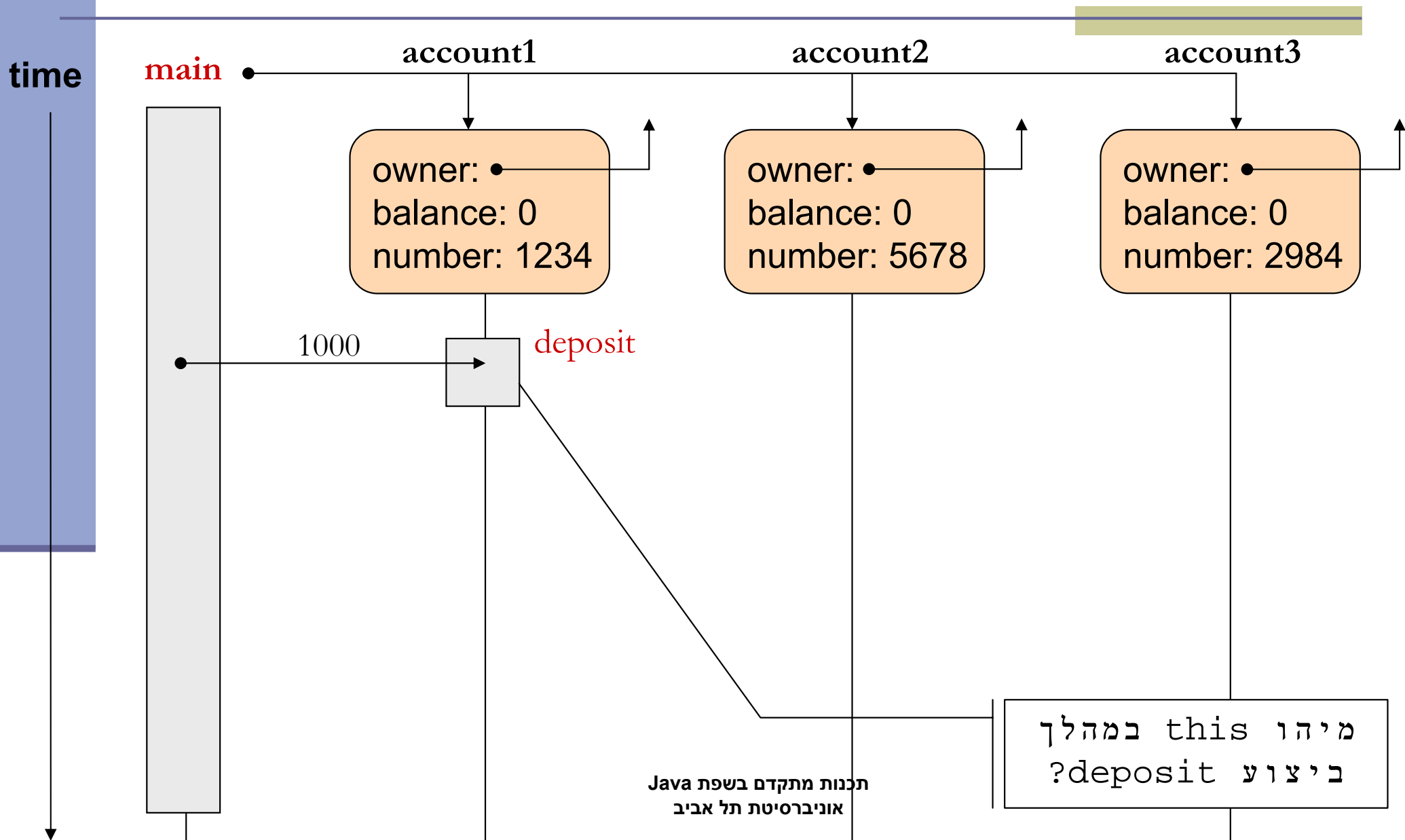
Object Diagram



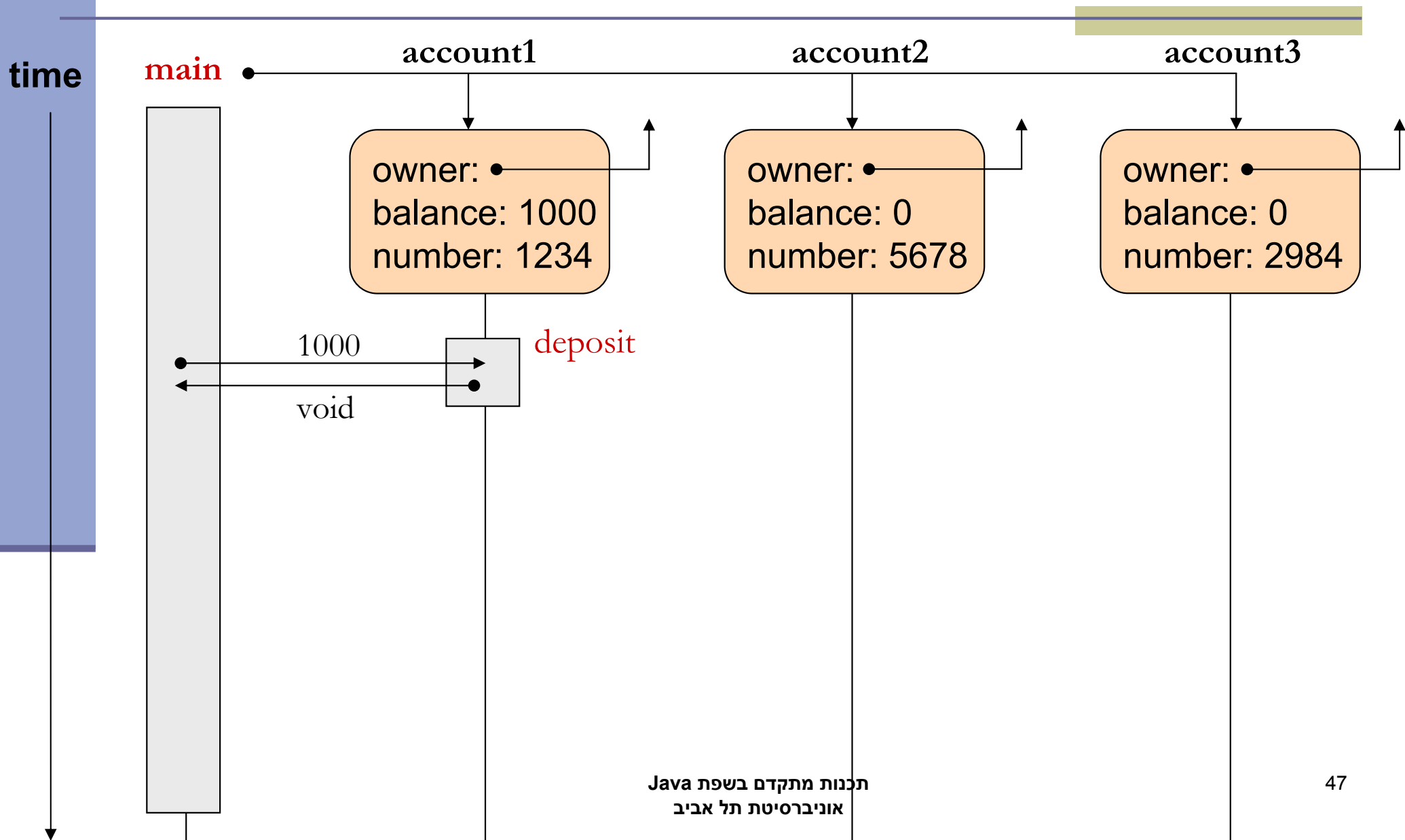
Message Sequence Chart

```
public class Bank {  
    public static void main(String[] args) {  
        Customer customer1 = new Customer("Avi Cohen", "025285244");  
        Customer customer2 = new Customer("Rita Stein", "024847638");  
  
        BankAccount account1 = new BankAccount(customer1, 1234);  
        BankAccount account2 = new BankAccount(customer2, 5678);  
        BankAccount account3 = new BankAccount(customer2, 2984);  
  
        → account1.deposit(1000);  
        account2.deposit(500);  
        account1.transferTo(100, account3);  
        account2.withdraw(300);  
  
        System.out.println("account1 has " + account1.getBalance());  
        System.out.println("account2 has " + account2.getBalance());  
    }  
}
```

Message Sequence Chart



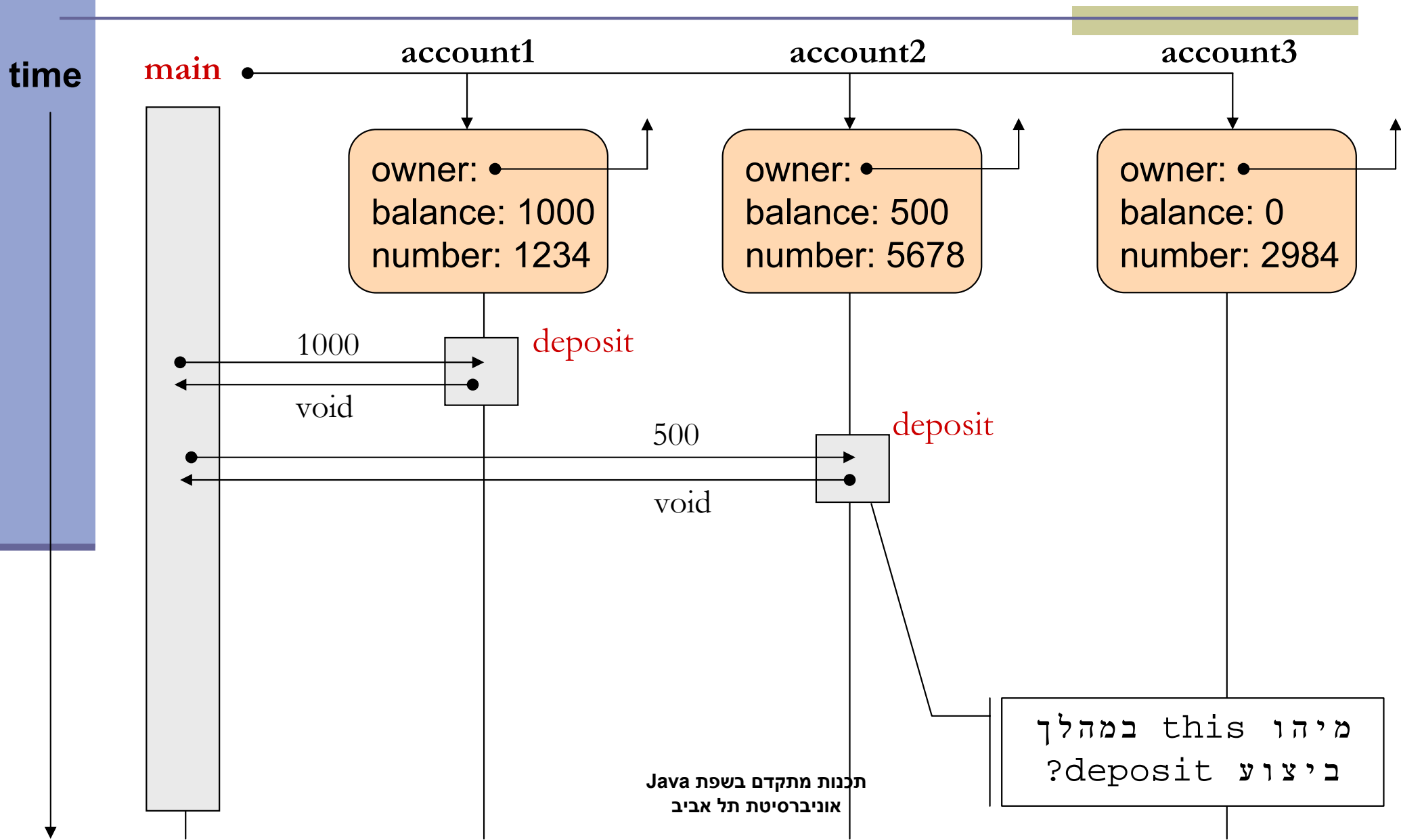
Message Sequence Chart



Message Sequence Chart

```
public class Bank {  
    public static void main(String[] args) {  
        Customer customer1 = new Customer("Avi Cohen", "025285244");  
        Customer customer2 = new Customer("Rita Stein", "024847638");  
  
        BankAccount account1 = new BankAccount(customer1, 1234);  
        BankAccount account2 = new BankAccount(customer2, 5678);  
        BankAccount account3 = new BankAccount(customer2, 2984);  
  
        account1.deposit(1000);  
        → account2.deposit(500);  
        account1.transferTo(100, account3);  
        account2.withdraw(300);  
  
        System.out.println("account1 has " + account1.getBalance());  
        System.out.println("account2 has " + account2.getBalance());  
    }  
}
```

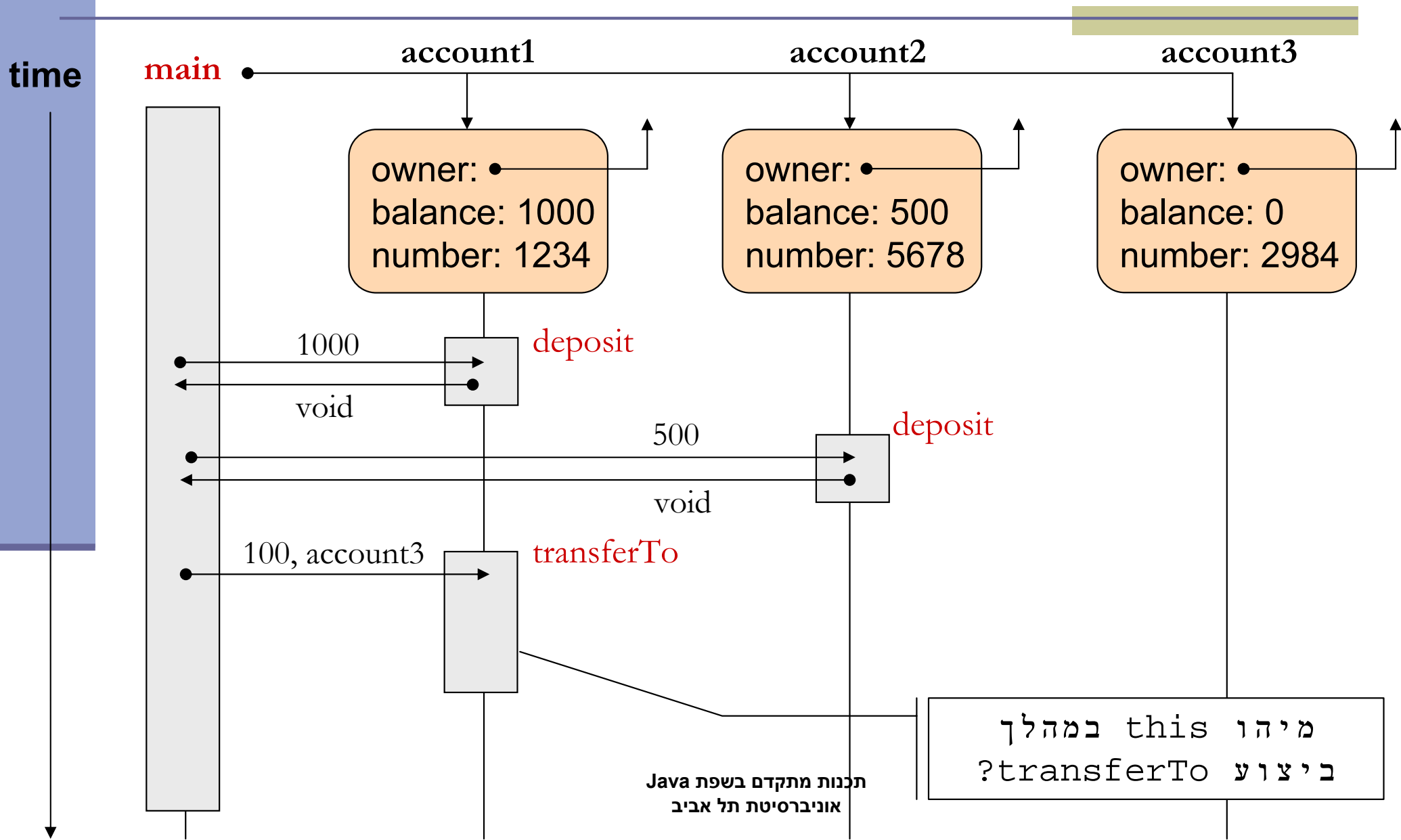

Message Sequence Chart



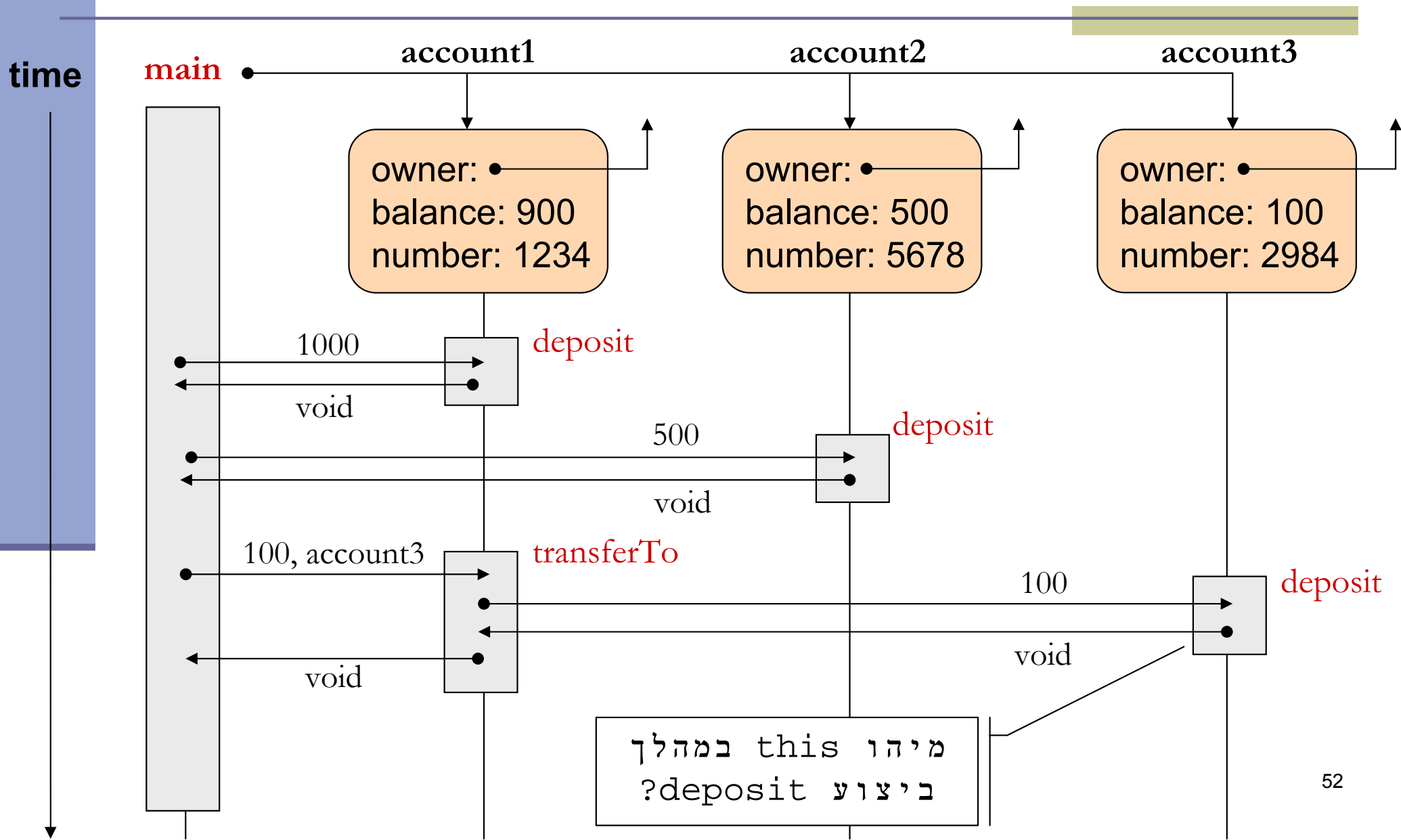
Message Sequence Chart

```
public class Bank {  
    public static void main(String[] args) {  
        Customer customer1 = new Customer("Avi Cohen", "025285244");  
        Customer customer2 = new Customer("Rita Stein", "024847638");  
  
        BankAccount account1 = new BankAccount(customer1, 1234);  
        BankAccount account2 = new BankAccount(customer2, 5678);  
        BankAccount account3 = new BankAccount(customer2, 2984);  
  
        account1.deposit(1000);  
        account2.deposit(500);  
        → account1.transferTo(100, account3);  
        account2.withdraw(300);  
  
        System.out.println("account1 has " + account1.getBalance());  
        System.out.println("account2 has " + account2.getBalance());  
    }  
}
```

Message Sequence Chart



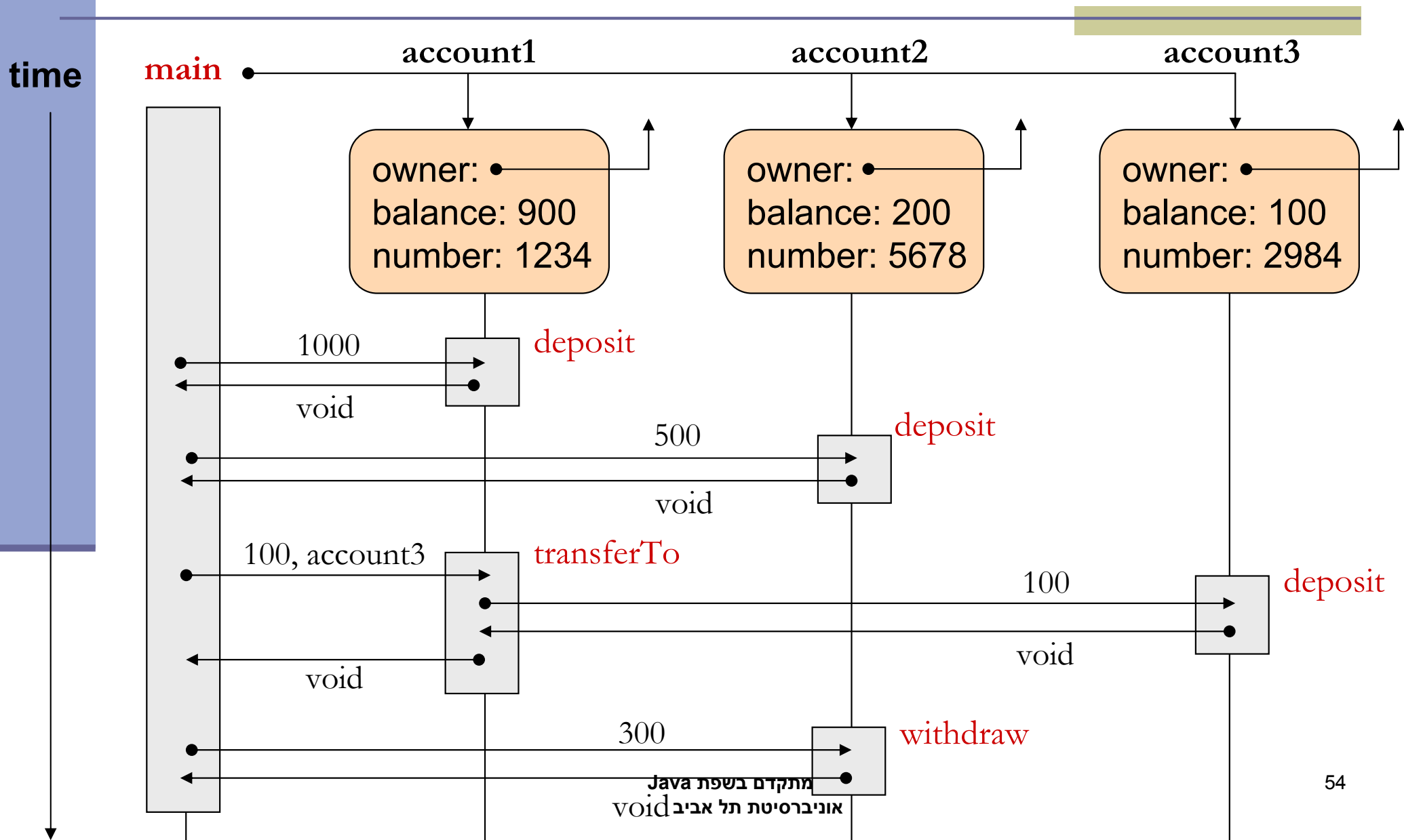
Message Sequence Chart



Message Sequence Chart

```
public class Bank {  
    public static void main(String[] args) {  
        Customer customer1 = new Customer("Avi Cohen", "025285244");  
        Customer customer2 = new Customer("Rita Stein", "024847638");  
  
        BankAccount account1 = new BankAccount(customer1, 1234);  
        BankAccount account2 = new BankAccount(customer2, 5678);  
        BankAccount account3 = new BankAccount(customer2, 2984);  
  
        account1.deposit(1000);  
        account2.deposit(500);  
        account1.transferTo(100, account3);  
        account2.withdraw(300);  
  
        System.out.println("account1 has " + account1.getBalance());  
        System.out.println("account2 has " + account2.getBalance());  
    }  
}
```

Message Sequence Chart



Output

```
public class Bank {  
    public static void main(String[] args) {  
        Customer customer1 = new Customer("Avi Cohen", "025285244");  
        Customer customer2 = new Customer("Rita Stein", "024847638");  
  
        BankAccount account1 = new BankAccount(customer1, 1234);  
        BankAccount account2 = new BankAccount(customer2, 5678);  
        BankAccount account3 = new BankAccount(customer2, 2984);  
  
        account1.deposit(1000);  
        account2.deposit(500);  
        account1.transferTo(100, account3);  
        account2.withdraw(300);  
  
        System.out.println("account1 has " + account1.getBalance());  
        System.out.println("account2 has " + account2.getBalance());  
    }  
}
```

output: account1 has 900.0
account2 has 200.0