IMPLEMENTING CLASSES

Chapter 3

Black Box

- Something that magically does its thing!
- You know what it does but not how.
- You really don't care how.
- Example – car
- Its interaction with the outside world is know.
- You know how to interface

Encapsulation

- The hiding of unimportant details
- Black box provides encapsulation
- Somebody has to come up with the concept.
- Software uses encapsulation to take complex routines and form “black boxes”
- Object-oriented programming
  - Black-boxes from which a program is manufactured are called objects
  - We learn the interfaces and what they are to do but not how they do it.

Abstraction

- Taking away inessential features
- Formal definition: the process of finding the essential feature set for a building block of a program such as a class
- What we know
  - What it does
  - How to interface with it
- What we don't how
  - How it does it

Classes

- Designer must understand the problem
- Designer must understand the behavior of the class
- Others can use the class
  - They don't need to understand the workings
- Must provide a means of interfacing with class

Example

- We will use a bank account
- We will call our class BankAccount
- This example will be used through out the semester and this book

What to Do

- Design a BankAccount class that other programmers can use (abstraction)
- Find essential operations
  - Deposit money
  - Withdraw money
  - Get the current balance
- Programmers who use class will view its objects as black boxes

What To Do

- Each operation == a method
- Turn the essential operations into a method or a black box

What To Do

- Methods needed
  - public void deposit(double amount)
  - public void withdraw(double amount)
  - public double getBalance()
- Which are accessors?
- Which are mutators?

What to Do

- When we want to use one of the methods we must call it.
- Example of methods calls
  - harrysChecking.deposit(2000)
  - harrysChecking.withdraw(500)
  - System.out.println(harryChecking.getBalance())

Methods

- Every method contains:
  - An access specifier (usually public)
  - The return type
    - Void (no return)
    - Type (int, double, String)
  - Name of the method
  - List of the parameters () or (something)

Methods

```
public void deposit(double amount)
{
        method body
}
public void withdraw(double amount)
public double getBalance( )
```

Contructors

- Contain instructions to initialize objects
- Resemble methods
- When you create an object the constructor is called

```
BankAccount harrysChecking = new BankAccount ();
BankAccount harrysChecking = new BankAccount (5000);
```

Constructor

- Difference between constructor and method
  - Name of constructor is the same as the class
  - Have not return type not even void

Creating Constructors

```
public class BankAccount
{
        // Constructors
        public BankAccount()
        {
        Fill in later
        }
        public BankAccount(double initialBalance)
        {
        Fill in later
        }
}
```

Instance Field

- An object stores its data in instance fields
- Instance fields are the variables associated with the object
- Field – storage location within a block of memory
- Instance – the object of the class

Instance Field

- Instance field declaration consists of:
  - An access specifier (usually private)
  - The type of the instance field
  - Name of the instance field

Instance Field Declaration

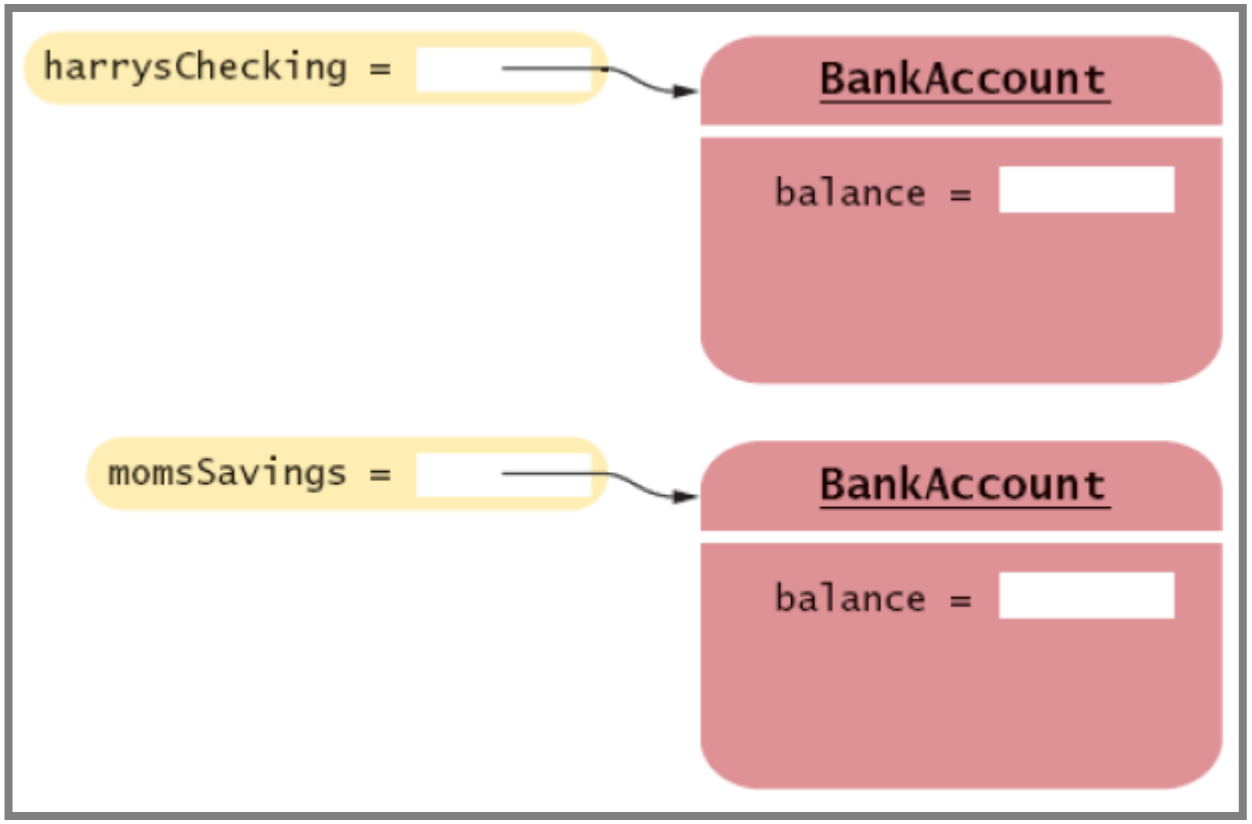
```
public class BankAccount
{
        …..
        private double balance;
        …..
}
```

Every object of BankAccount will have a field named balance

What Does Private Mean

- Instance fields are hidden from the programmer who uses the class
- They are only of concern to the programmer who implement the class
- All access must occur through public methods not instance fields
- The process of hiding data and providing methods for data access is called encapsulation

Instance Fields



Access to Instance Fields

- Separate method
- Example: getBalance
  - Returns the balance
  - Can grant the user access to only get the balance the not change the balance
  - You control who has access to what

Implementing Constructors

- Constructors contain instructions to initialize the instance fields of an object

```
public BankAccount()
{
   balance = 0;
}
public BankAccount(double initialBalance)
{
   balance = initialBalance;
}
```

Constructor Call Example

```
BankAccount harrysChecking = new BankAccount(1000.0);
```

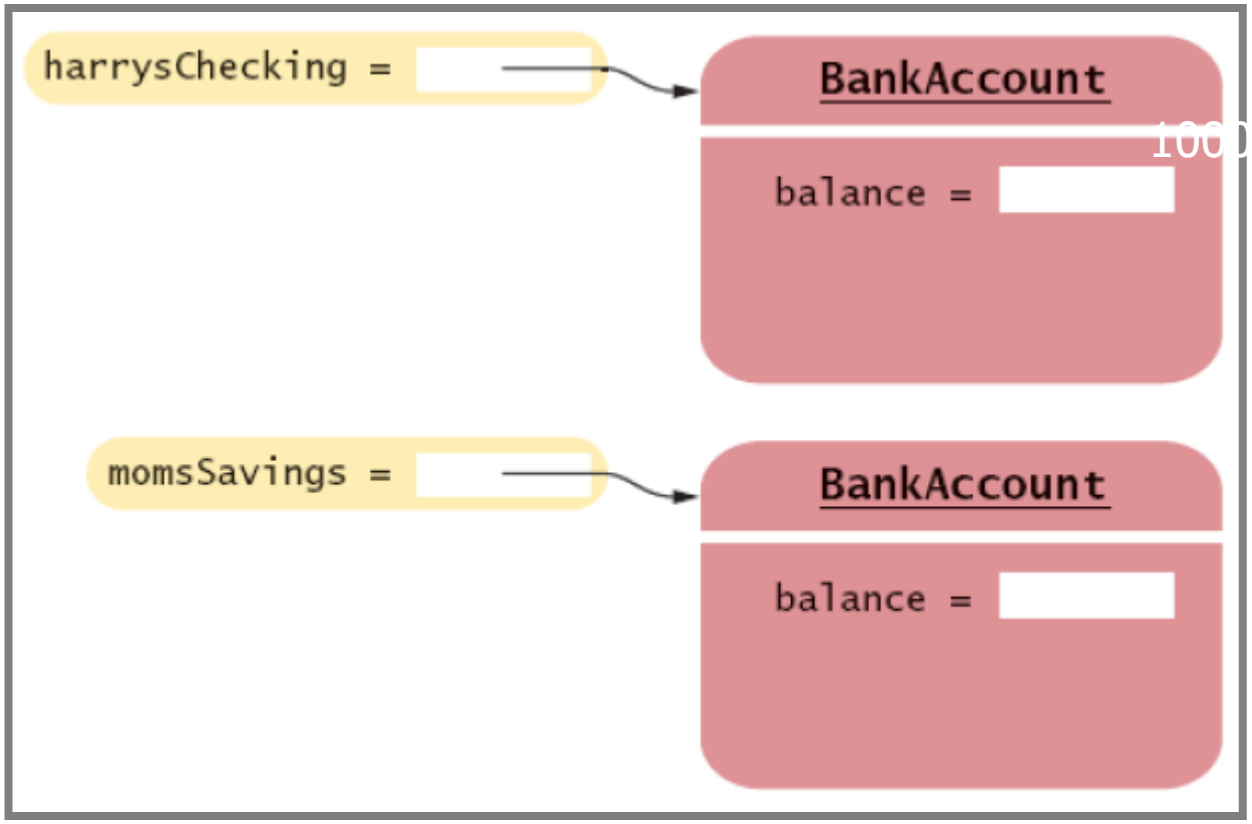
- Create a new object of `type BankAccount`
- Call the 2nd constructor (since the construction parameter supplied matches the type of 2nd)
- Set the parameter variable `initialBalance` to 1000
- In the constructor the `balance` instance field of the newly created object is set to `initialBalance`

Constructor Call Example

```
BankAccount harrysChecking = new BankAccount(1000);
```

- Return an object reference, that is, the memory location of the object, as the value of the `new` expression
- Store that object reference in the `harrysChecking` variable

Instance Fields



File BankAccount.java

```
01: /**
02:    A bank account has a balance that can be changed by
03:    deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:    /**
08:       Constructs a bank account with a zero balance.
09:    */
10:    public BankAccount()
11:    {
12:       balance = 0;
13:    }
14:
15:    /**
16:       Constructs a bank account with a given balance.
17:       @param initialBalance the initial balance
18:    */
```

Continued…

File BankAccount.java

```
19:    public BankAccount(double initialBalance)
20:    {
21:       balance = initialBalance;
22:    }
23:
24:    /**
25:       Deposits money into the bank account.
26:       @param amount the amount to deposit
27:    */
28:    public void deposit(double amount)
29:    {
30:       double newBalance = balance + amount;
31:       balance = newBalance;
32:    }
33:
34:    /**
35:       Withdraws money from the bank account.
36:       @param amount the amount to withdraw
```

Continued…

File BankAccount.java

```
37:     */
38:     public void withdraw(double amount)
39:     {
40:        double newBalance = balance - amount;
41:        balance = newBalance;
42:     }
43:
44:     /**
45:        Gets the current balance of the bank account.
46:        @return the current balance
47:     */
48:     public double getBalance()
49:     {
50:        return balance;
51:     }
52:
53:     private double balance;
54:
```

Testing a Class

- We need to be sure our Class (in this case BankAccount) works correctly
- Write a test case or tester class
  - Construct one or more objects of the class that is being tested
  - Invoke one or more methods
  - Print out one or more results
  - Print the expected results

BankAccountTester.java

```
01: /**
02:    A class to test the BankAccount class.
03: */
04: public class BankAccountTester
05: {
06:    /**
07:       Tests the methods of the BankAccount class.
08:       @param args not used
09:    */
10:    public static void main(String[] args)
11:    {
12:       BankAccount harrysChecking = new BankAccount();
13:       harrysChecking.deposit(2000);
14:       harrysChecking.withdraw(500);
          System.out.println("Expected result: 1500");
15:       System.out.println(harrysChecking.getBalance());
16:    }
17:
```

Categories of Variables

- Categories of variables
  - Instance fields (`balance` in `BankAccount`)
  - Local variables (`newBalance` in `deposit` method)
  - Parameter variables (`amount` in `deposit` method)
- An instance field belongs to an object
  - The fields stay alive until no method uses the object any longer

Categories of Variables

- Local & parameter variable
  - Local variables must be initialized
  - Parameter variables are initialized in the method call
- Instance fields that are numbers are initialized to zero by default
- Object references are set to “null” by default

Categories of Variables

- In Java, the *garbage collector* periodically reclaims objects when they are no longer used
- Local and parameter variables belong to a method

Lifetime of Variables

```
harrysChecking.deposit(500);
double newBalance = balance + amount;
balance = newBalance;
```

Continued…

Lifetime of Variables

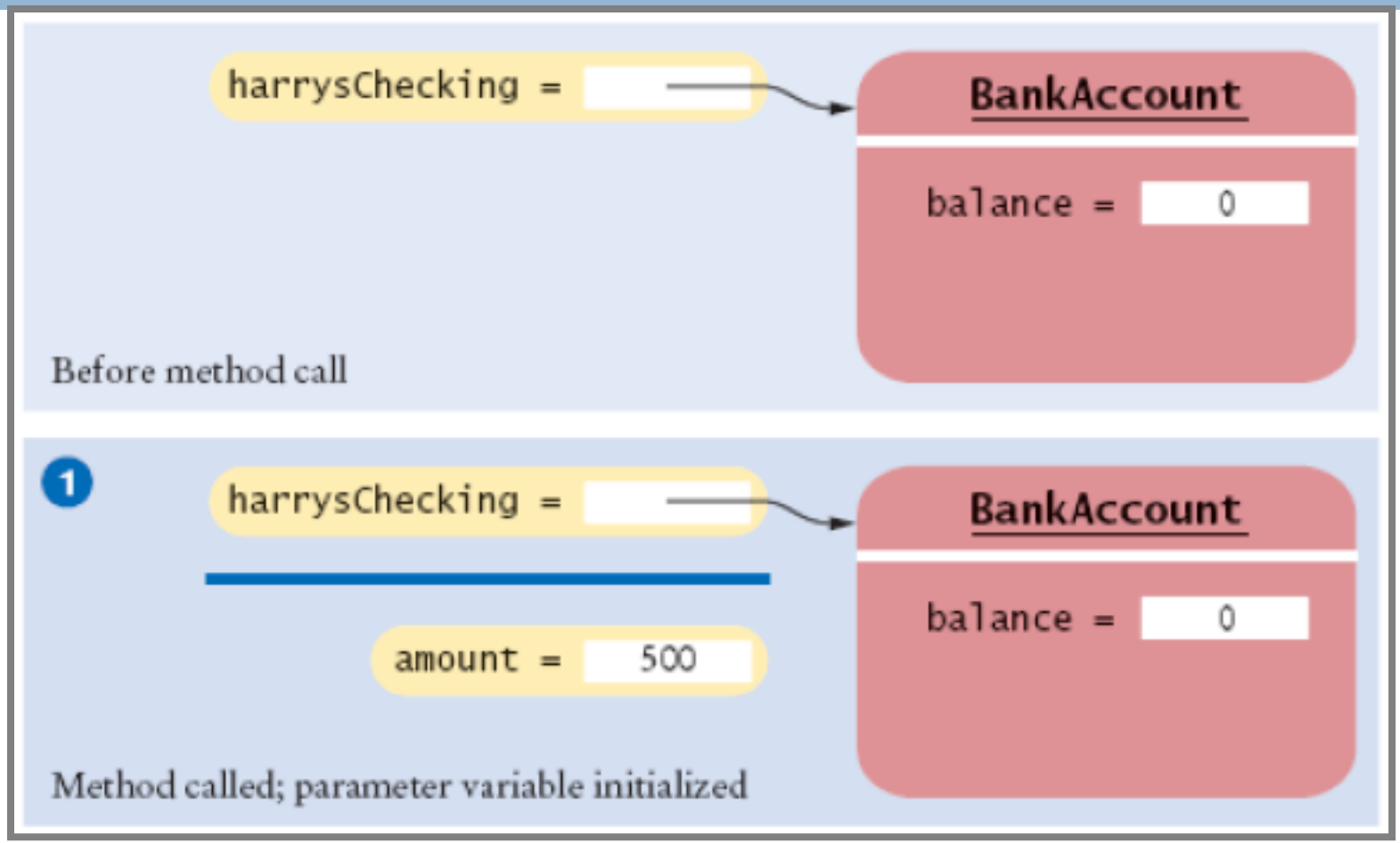


Figure 7:
Lifetime of Variables

Continued…

Lifetime of Variables

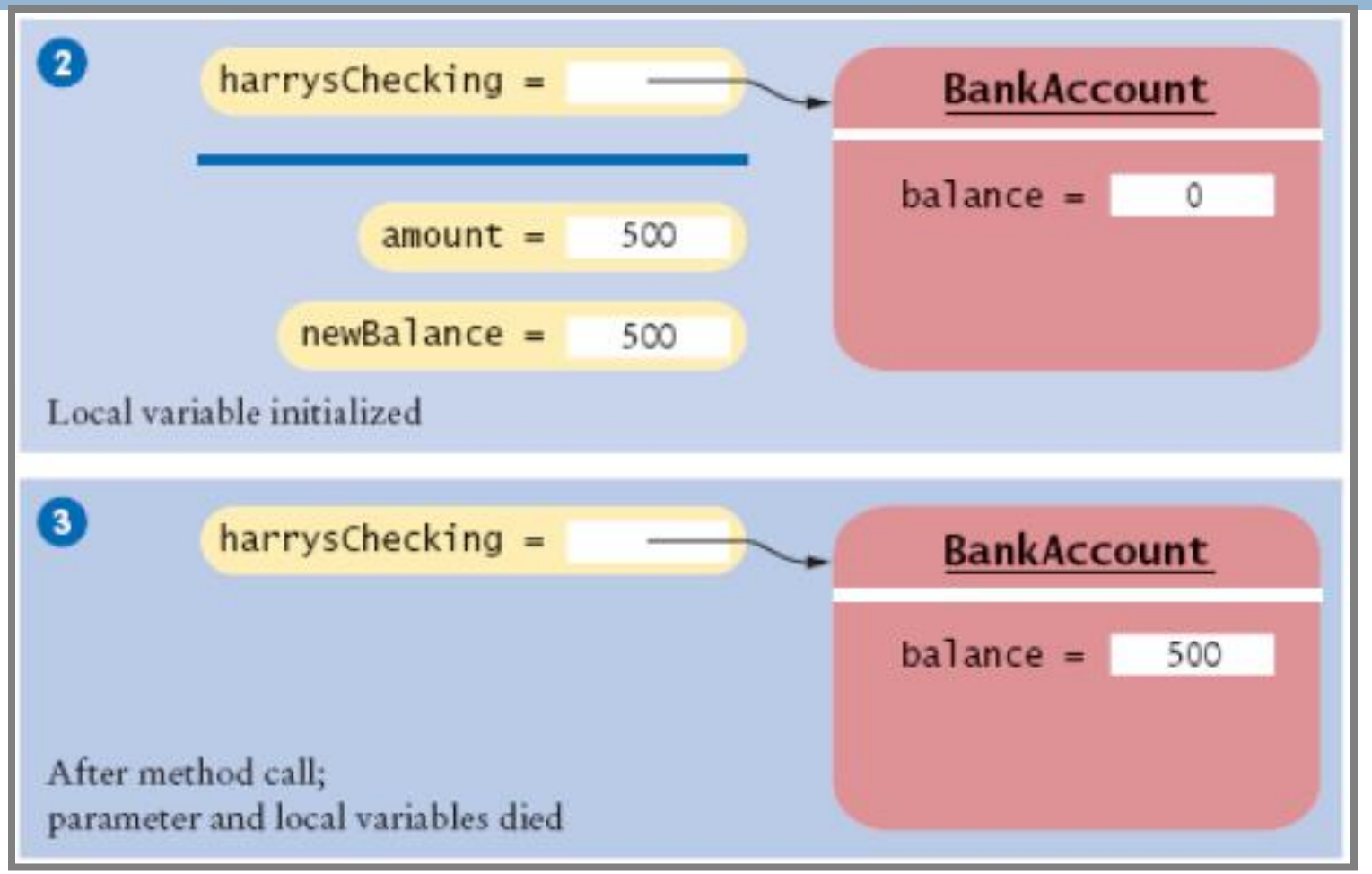


Figure 7:
Lifetime of Variables

Implicit and Explicit Method Parameters

- The implicit parameter of a method is the object on which the method is invoked

```
momsSavings.withdraw(500);
```

- Sometime you will see the word `this` used as a reference and denotes the implicit parameter

Implicit and Explicit Method Parameters

- Use of an instance field name in a method denotes the instance field of the implicit parameter

```
public void withdraw(double amount)
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Implicit and Explicit Method Parameters

```
momsSavings.withdraw(500)
```

```
double newBalance = momsSavings.balance - amount;
momsSavings.balance = newBalance;
```

Implicit Parameters and `this`

- Every method has one implicit parameter
- The implicit parameter is always called `this`
- Exception: Static methods do not have an implicit parameter (more later) – remember main (no object)

```
double newBalance = balance + amount;
// actually means
double newBalance = this.balance + amount;
```

Implicit Parameters and this