



# D06

# PROGRAMMING with JAVA

## Ch3 – Implementing Classes

PowerPoint presentation, created by Angel A. Juan - [ajuanp@gmail.com](mailto:ajuanp@gmail.com),  
for accompanying the book "Big Java", by Cay S. Horstmann (2006)

# Chapter Goals

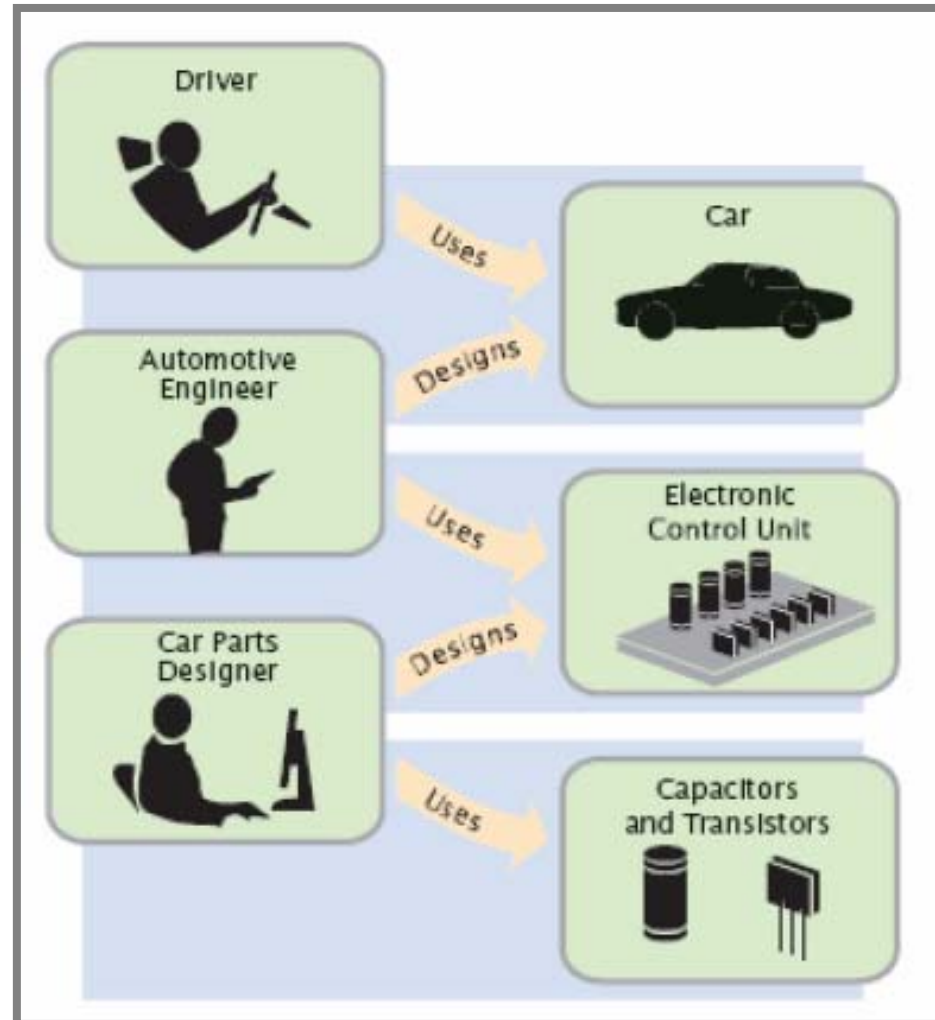
---

- To become familiar with the process of implementing classes
- To be able to implement simple methods
- To understand the purpose and use of **constructors**
- To understand how to access **instance fields** and local variables
- To appreciate the importance of **documentation comments**

# Black Boxes

- A black box provides **encapsulation**, the hiding of unimportant details
- Black boxes in a car: transmission, electronic control module, etc.

Figure 1: Levels of Abstraction in Automobile Design



*Continued...*

# Levels of Abstraction: Software Design

---

- Problem: in old times, computer programs manipulated **primitive types** such as numbers and characters; manipulating too many of these primitive quantities is too much for programmers and leads to errors
- Solution: to **encapsulate** routine computations into software black boxes
- In **object-oriented programming** the black boxes from which a program is manufactured are called **objects**
- **Encapsulation**: programmer using an object knows about its behavior, but not about its internal structure
- **Abstraction** used to invent higher-level data types

*Continued...*

# Levels of Abstraction: Software Design

- In software design, you can design good and bad abstractions with equal facility
- First, define behavior of a class; then, **implement** it

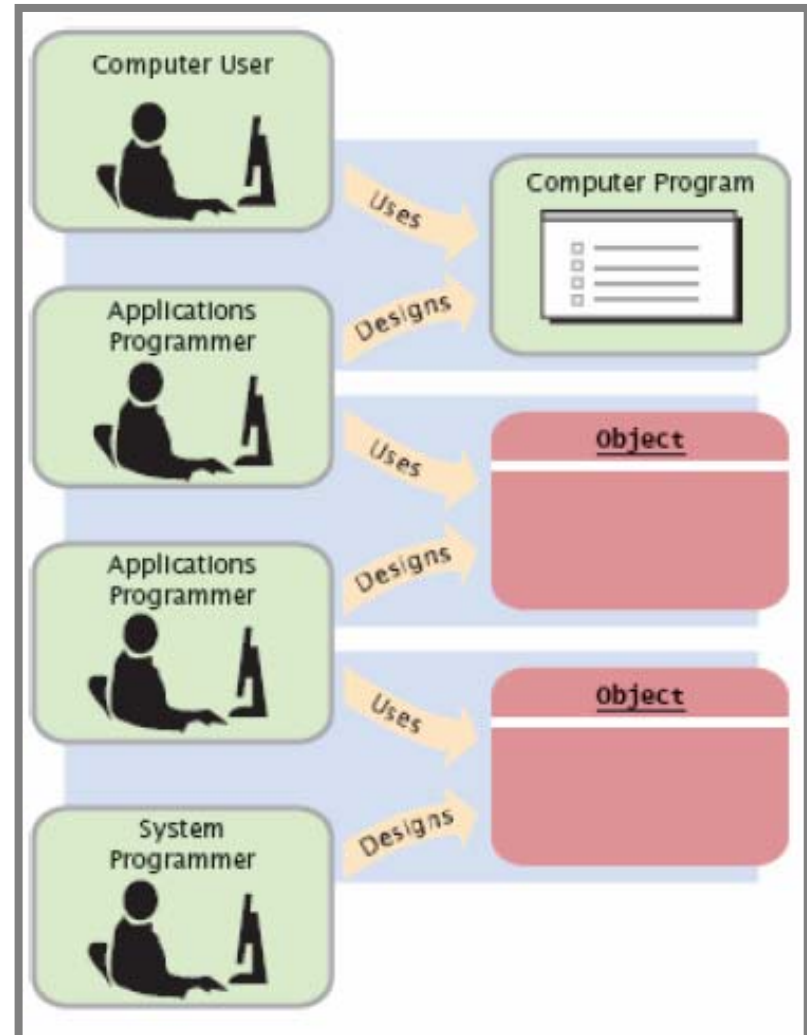


Figure 2: Levels of Abstraction  
in Software Design

# Self Check

---

1. In Chapters 1 and 2, you used `System.out` as a black box to cause output to appear on the screen. Who designed and implemented `System.out`?
2. Suppose you are working in a company that produces personal finance software. You are asked to design and implement a class for representing bank accounts. Who will be the users of your class?

# Answers

---

1. The programmers who designed and implemented the Java library
2. Other programmers who work on the personal finance application

# Designing the Public Interface of a Class: Methods

- Task: to design a **BankAccount** class that other programmers will use
- Essential behavior of a bank account (abstraction):
  - deposit money
  - withdraw money
  - get balance
- Methods of the **BankAccount** class:

```
deposit()  
withdraw()  
getBalance()
```



In Java, operations are expressed as **method calls**. We want to support method calls such as the following:

```
harrysChecking.deposit(2000);  
harrysChecking.withdraw(500);  
System.out.println(harrysChecking.getBalance());
```

mutator methods

accessor method

# Designing the Public Interface of a Class: Method Definition

- Every **method definition** contains the following parts:
  - An access specifier (usually **public**)
  - The return type (such as **String** or **void**)
  - The method name (such as **deposit**)
  - A list of parameters, including a type and a name for each parameter (such as **double amount** for the method **deposit**)
  - The method body, enclosed in braces { }



The **access specifier** controls which other methods can call this method. Most methods should be declared as **public** (that way, all other methods in a program can call them); **private** methods can only be called from other methods of the same class

# Syntax 3.1: Method Definition

```
accessSpecifier returnType methodName(parameterType  
parameterName, . . . )  
{  
    method body  
}
```

## **Examples:**

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

## **Purpose:**

To define the behavior of a method

# Designing the Public Interface of a Class: Constructor Definition

- A **constructor** initializes the instance variables when a new object is created: the statements in the constructor body will set the internal data of the object that is being constructed



A constructor is very similar to a method, with two important differences:

- **Always:** constructor name = class name
- Constructors have no return type (not even **void**)
- All constructors of a class have the same name; the compiler can tell them apart because they take different parameters

*Continued...*

# Designing the Public Interface of a Class: Constructor Definition

- When defining a class, you place all constructor and method definitions inside, like this:

```
public class BankAccount
{
    // Constructors
    public BankAccount()
    {
        // constructor body
    }
    public BankAccount(double initialBalance)
    {
        // constructor body
    }

    // Public methods

    // Private fields
}
```

- The public constructors and methods of a class form the **public interface** of the class

# Syntax 3.2: Constructor Definition

```
accessSpecifier ClassName(parameterType parameterName, . . . )  
{  
    constructor body  
}
```

## **Example:**

```
public BankAccount(double initialBalance)  
{  
    . . .  
}
```

## **Purpose:**

To define the behavior of a constructor

# Syntax 3.3: Class Definition

```
accessSpecifier class ClassName
{
    constructors
    methods
    fields
}
```

## Example:

```
public class BankAccount
{
    public BankAccount(double initialBalance) { . . . }
    public void deposit(double amount) { . . . }
    . . .
}
```

## Purpose:

To define a class, its public interface, and its implementation details

# Self Check

---

3. How can you use the methods of the public interface to *empty* the `harrysChecking` bank account?
4. Suppose you want a more powerful bank account abstraction that keeps track of an *account number* in addition to the balance. How would you change the public interface to accommodate this enhancement?

# Answers

---

3. `harrysChecking.withdraw(harrysChecking.getBalance())`

4. Add an `accountNumber` parameter to the constructors, and add a `getAccountNumber` method. There is no need for a `setAccountNumber` method—the account number never changes after construction.

# Commenting the Public Interface

---

- When programming in Java, provide **documentation comments** for every class and every public method



Follow these steps to add documentation comments:

- Before the class or public method definition, start the documentation comment with **/\*\***
- Describe the class or method's purpose
- If a method, for each parameter supply a line that starts with **@param** followed by the parameter name and a short explanation
- If a method, supply a line that starts with **@return** describing the return value
- End the documentation comment with **\*/**
- Use the **javadoc** program to automatically generate a neat set of HTML pages with the documentation comments

*Continued...*

# Commenting the Public Interface

```
/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
public class BankAccount
{
    /**
     * Withdraws money from the bank account.
     * @param the amount to withdraw
     */
    public void withdraw(double amount)
    {
        ...
    }

    /**
     * Gets the current balance of the bank account.
     * @return the current balance
     */
    public double getBalance()
    {
        ...
    }

    ...
}
}
```

# Javadoc Method Summary & Detail



Figure 3:  
A Method Summary  
Generated by javadoc

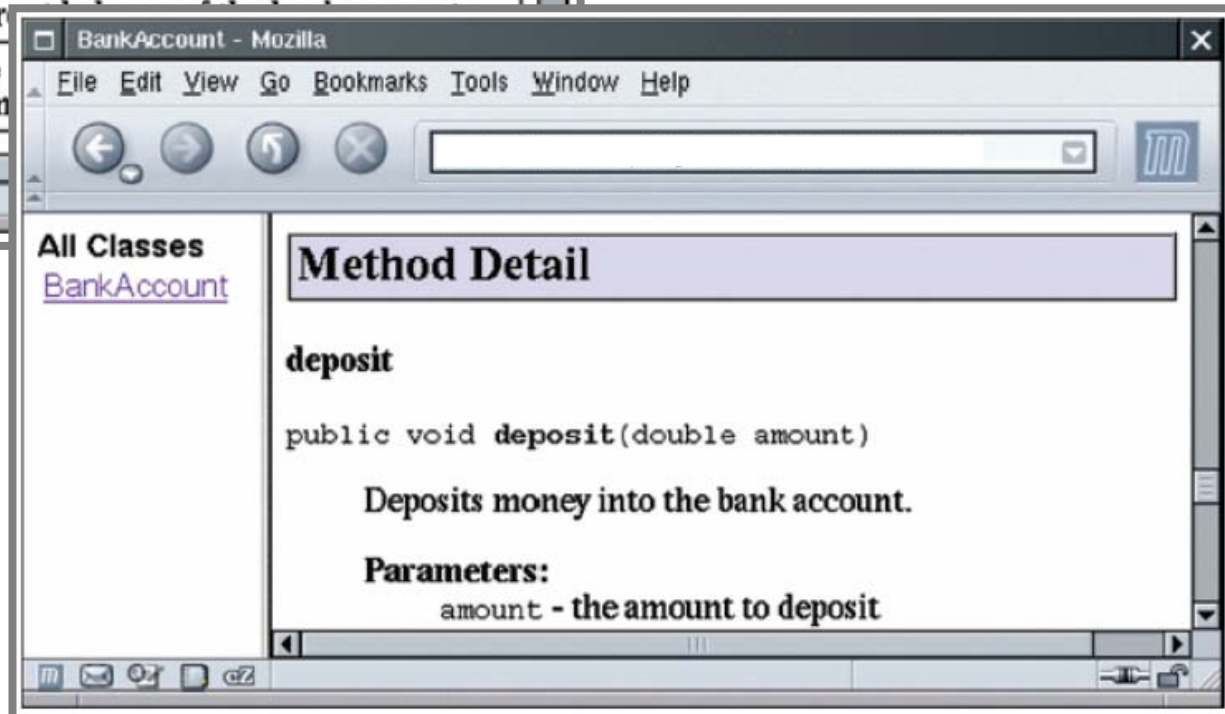


Figure 4:  
Method Detail  
Generated by javadoc

# Self Check

---

5. Suppose we enhance the `BankAccount` class so that each account has an account number. Supply a documentation comment for the constructor `BankAccount(int accountNumber, double initialBalance)`
6. Why is the following documentation comment questionable?

```
/**
 * Each account has an account number.
 * @return the account number of this account.
 */
int getAccountNumber()
```

# Answers

---

5. 

```
/**  
    Constructs a new bank account with a given initial balance.  
    @param accountNumber the account number for this account  
    @param initialBalance the initial balance for this account  
*/
```
6. The first sentence of the method description should describe the method—it is displayed in isolation in the summary table

# Instance Fields

- A **field** is a technical term for a storage location inside a block of memory. An **instance** of a class is an object of the class
- An object stores its data in **instance fields** or **instance variables** (a storage location that is present in each object of a class)
- The class declaration specifies the instance fields:

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

 You should declare all instance fields as **private**

*Continued...*

# Instance Fields

- An **instance field declaration** consists of:
  - An access specifier (usually **private**)
  - The type of the instance field (such as `double`)
  - The name of the instance field (such as `balance`)

 Each object of a class has its own set of instance fields

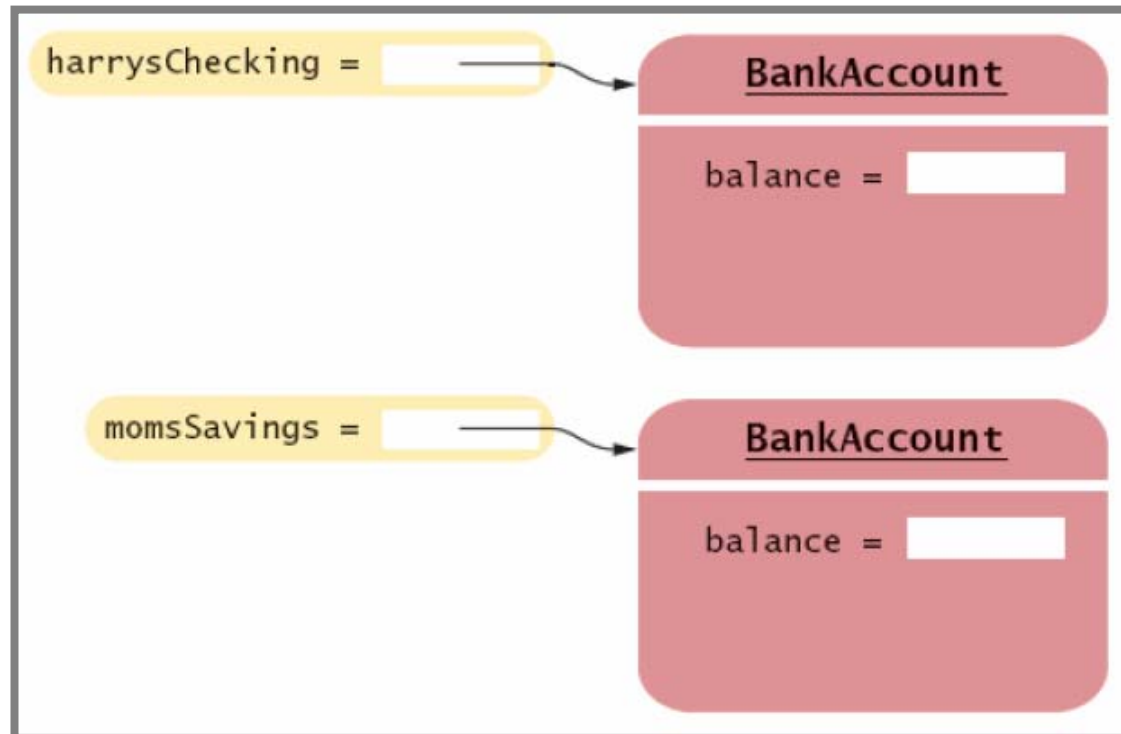


Figure 5:  
Instance Fields

# Syntax 3.4: Instance Field Declaration

```
accessSpecifier class ClassName
{
    . . .
    accessSpecifier fieldType fieldName;
    . . .
}
```

## **Example:**

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

## **Purpose:**

To define a field that is present in every object of a class

# Accessing Instance Fields

- Instance fields use to be declared with the access specifier **private**, i.e.: they can be accessed only by the methods of the same class
- The `deposit()` method of the `BankAccount` class can access the `private` instance field, others cannot:

```
public void deposit(double amount)
{
    double newBalance = balance + amount;
    balance = newBalance;
}
```

```
public class BankRobber
{
    public static void main(String[] args)
    {
        BankAccount momsSavings = new BankAccount(1000);
        . . .
        momsSavings.balance = -1000; // ERROR
    }
}
```



The process of hiding the data and providing methods for data access is called **encapsulation**

# Self Check

---

7. Suppose we modify the `BankAccount` class so that each bank account has an account number. How does this change affect the instance fields?
8. What are the instance fields of the `Rectangle` class?

# Answers

---

## 7. An instance field

```
private int accountNumber;
```

needs to be added to the class

## 8.

```
private int x;  
private int y;  
private int width;  
private int height;
```

# Implementing Constructors

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

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



## A Constructor Call:

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

- Create a new object of type `BankAccount`
- Call the second constructor (a construction parameter is supplied)
- Set the explicit parameter `initialBalance` to 1000
- Set the `balance` instance field of the newly created object to `initialBalance`
- Return an object reference as the value of the `new` expression
- Store that object reference in the `harrysChecking` variable

# Implementing Methods

- The **return** statement instructs the method to terminate and return an output to the statement that called it. Some methods do not return any value (**void**):

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

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



A Method Call:

```
harrysChecking.withdraw(500);
```

- Set the parameter variable `amount` to 500
- Fetch the `balance` field of the object whose location is stored in `harrysChecking`
- Subtracts the value of `amount` from `balance` and store the result in the variable `newBalance`
- Store the value of `newBalance` in the `balance` instance field, overwriting the old value

# Syntax 3.5: The `return` Statement

```
return expression;  
or  
return;
```

## **Example:**

```
return balance;
```

## **Purpose:**

To specify the value that a method returns, and exit the method immediately. The return value becomes the value of the method call expression.

# 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: }
```

# Self Check

---

9. How is the `getWidth` method of the `Rectangle` class implemented?
10. How is the `translate` method of the `Rectangle` class implemented?

# Answers

9.

```
public int getWidth()  
{  
    return width;  
}
```

10. There is more than one correct answer. One possible implementation is as follows:

```
public void translate(int dx, int dy)  
{  
    int newx = x + dx;  
    x = newx;  
    int newy = y + dy;  
    y = newy;  
}
```

# Testing a Class

---

- A **test class** is a class with a `main( )` method that contains statements to test another class. It typically carries out the following steps:
  1. Construct one or more objects of the class that is being tested
  2. Invoke one or more methods
  3. Print out one or more results



When building a program composed by (a) one class, and (b) its associated test class, you usually need to:

1. Make a new subfolder for your program
2. Make two files, one for each class
3. Compile both files
4. Run the test program

*Continued...*

# File BankAccountTester.java

```
01: /**
02:     A class to test the BankAccount class.
03: */
04: public class BankAccountTeste
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);
15:         System.out.println(harrysChecking.getBalance());
16:     }
17: }
```

# Self Check

---

11. When you run the `BankAccountTester` program, how many objects of class `BankAccount` are constructed? How many objects of type `BankAccountTester`?
12. Why is the `BankAccountTester` class unnecessary in development environments that allow interactive testing, such as BlueJ?

# Answers

---

11. One `BankAccount` object, no `BankAccountTester` object. The purpose of the `BankAccountTester` class is merely to hold the main method
12. In those environments, you can issue interactive commands to construct `BankAccount` objects, invoke methods, and display their return values

# Categories of Variables

- We've seen three categories of variables
  - **Instance fields** (balance in BankAccount)
  - **Local variables** (newBalance in withdraw() method)
  - **Parameter variables** (amount in withdraw() method)



An instance field belongs to an object. Each object has its own copy of each instance field. Local and parameter variables belong to a method

- **Lifetime** of each variable category:
  - Instance fields: when an object is constructed, its instance fields are created. The fields stay alive until no method uses the object any longer (the JVM contains an agent called a **garbage collector** that periodically reclaims objects when they are no longer used)
  - Local and parameter variables: when their associated method runs, these variables come to life; when the method exits, they die immediately

*Continued...*

# Categories of Variables

---

- **Initialization** of each variable category:
  - You must initialize all local variables
  - Parameter variables are initialized with the values that are supplied in the method call
  - Instance fields are initialized with a default value if you don't explicitly set them in a constructor: instance fields that are numbers are initialized to **0**; object references are set to **null** (if an object reference is **null**, then it refers to no object at all)

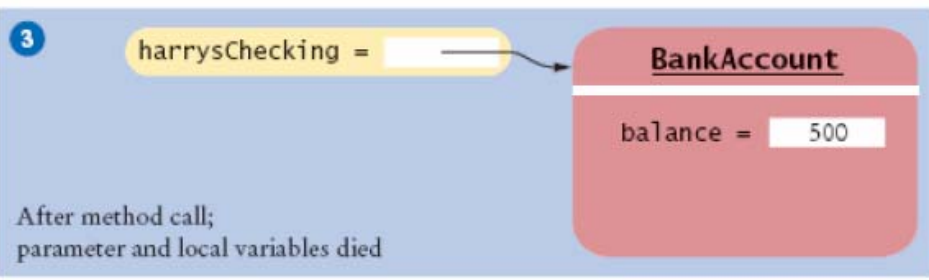
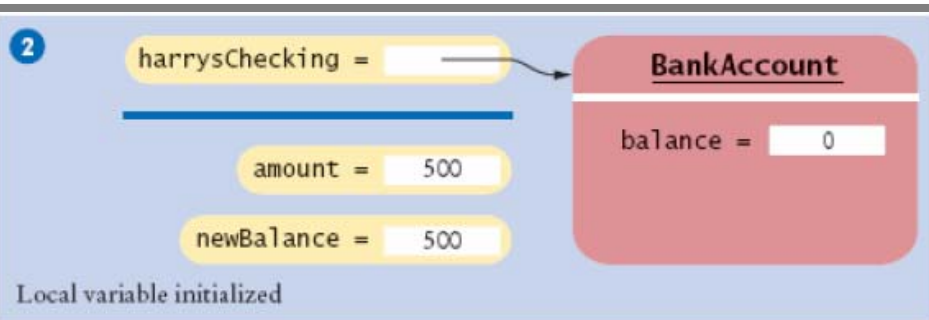
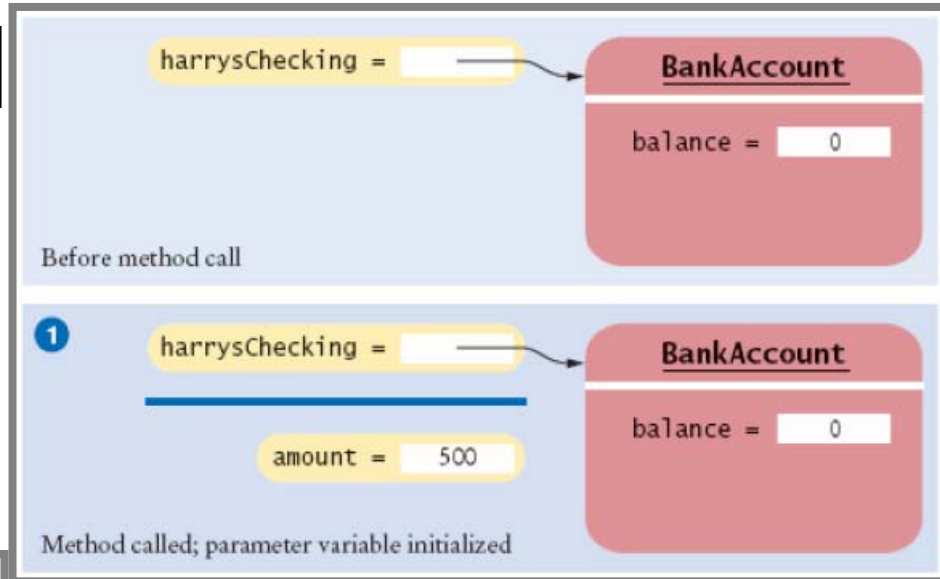


You should initialize every instance field explicitly in every constructor

# Lifetime of Variables

```
harrysChecking.deposit(500);
```

Figure 7:  
Lifetime of Variables



```
double newBalance = balance + amount;
```

```
balance = newBalance;
```

# Self Check

---

13. What do local variables and parameter variables have in common? In which essential aspect do they differ?
14. During execution of the `BankAccountTester` program in the preceding section, how many instance fields, local variables, and parameter variables were created, and what were their names?


# Answers

---

13. Variables of both categories belong to methods—they come alive when the method is called, and they die when the method exits. They differ in their initialization. Parameter variables are initialized with the call values; local variables must be explicitly initialized
14. One instance field, named `balance`. Three local variables, one named `harrysChecking` and two named `newBalance` (in the `deposit` and `withdraw` methods); two parameter variables, both named `amount` (in the `deposit` and `withdraw` methods)

# Implicit and Explicit Method Parameters

- The **implicit parameter** of a method is the object on which the method is invoked; the **explicit parameters** of a method are inputs enclosed in parentheses

 When you refer to an instance field inside a method, it means the instance field of the object on which the method was called:

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

`balance` is the balance of the object to the left of the dot:

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

means:

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

*Continued...*

# Implicit and Explicit Method Parameters

- If you need to, you can access the implicit parameter with the keyword **this**



Every method has one implicit parameter. It is always called **this** (exception: **static** methods do not have an implicit parameter)

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

- In contrast, methods can have any number of explicit parameters –or no explicit parameters at all

*Continued...*

# Implicit Parameters and `this`



When you refer to an instance field in a method, the compiler automatically applies it to the `this` parameter (some programmers actually prefer to manually insert the `this` parameter before every instance field):

```
momsSavings.deposit(500);
```

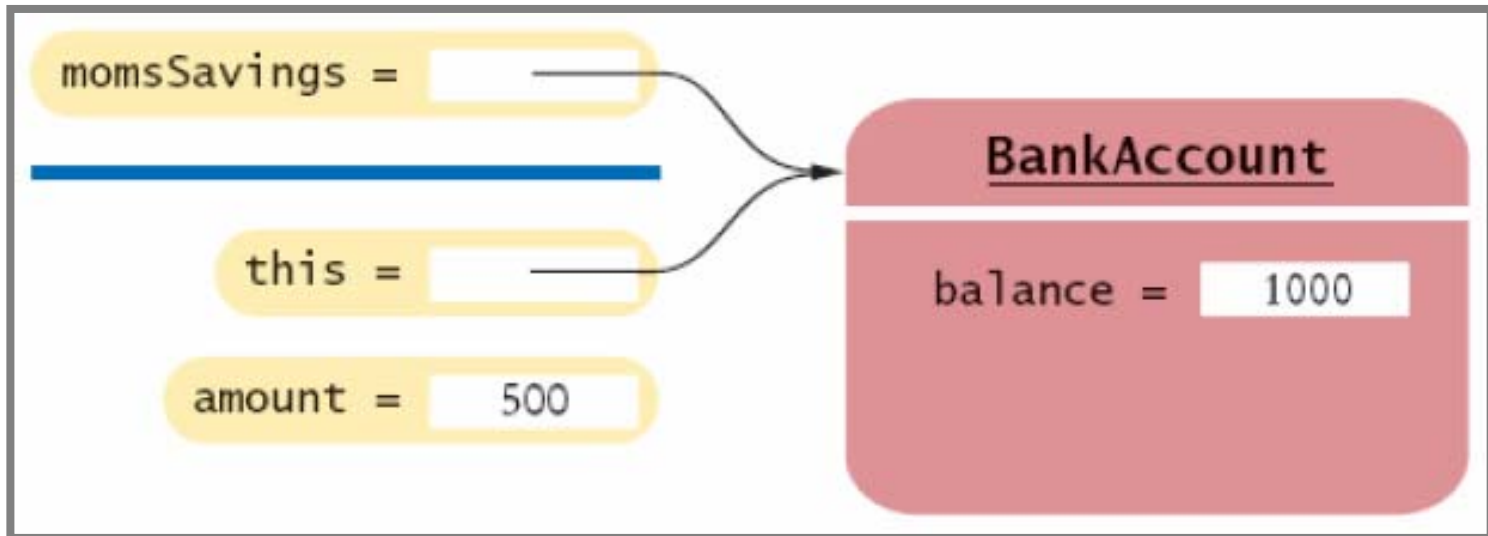


Figure 8: The Implicit Parameter of a Method Call

# Calling One Constructor from Another



Normally, **this** denotes a reference to the implicit parameter, but if **this** is followed by parentheses, it denotes a call to another constructor of this class (e.g., the command **this(0);** means “call another constructor of this class and supply the value 0”; such a constructor call can occur only as the first line in another constructor):

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

# Self Check

---

15. How many implicit and explicit parameters does the `withdraw` method of the `BankAccount` class have, and what are their names and types?
16. In the `deposit` method, what is the meaning of `this.amount`? Or, if the expression has no meaning, why not?
17. How many implicit and explicit parameters does the `main` method of the `BankAccountTester` class have, and what are they called?

# Answers

---

15. One implicit parameter, called `this`, of type `BankAccount`, and one explicit parameter, called `amount`, of type `double`
16. It is not a legal expression. `this` is of type `BankAccount` and the `BankAccount` class has no field named `amount`
17. No implicit parameter—the method is static—and one explicit parameter, called `args`

# Chapter Summary

---

- A **method definition** contains an access specifier (usually public), a return type, a method name, parameters, and the method body
- **Constructors** contain instructions to initialize objects. The constructor name is always the same as the class name
- Provide **documentations comments** for every class, every method, every parameter and every return value
- An object uses **instance fields** to store its state –the data it needs to execute its methods
- Each object of a class has its own set of instance fields
- You should declare all instance fields as `private`
- **Encapsulation** is the process of hiding object data and providing methods for data access

*Continued...*

# Chapter Summary

---

- **Constructors** contain instructions to initialize the instance fields of an object
- Use the **return** statement to specify the value that a method returns to its caller
- To test a class, write a second class to execute test instructions
- Instance fields belong to an object. Parameter variables and local variables belong to a method –they die when the method exits
- Instance fields are initialized to a default value, but you must initialize local variables
- The **implicit parameter** of a method is the object on which the method is invoked. The **this** reference denotes it
- Use of an instance field name in a method denotes the instance field of the implicit parameter