Objective

Implementing Classes:

- Process of implementing classes
- Implement simple methods
- Constructors
- Instance fields and local variables
- Documentation & Comments

Black Box:

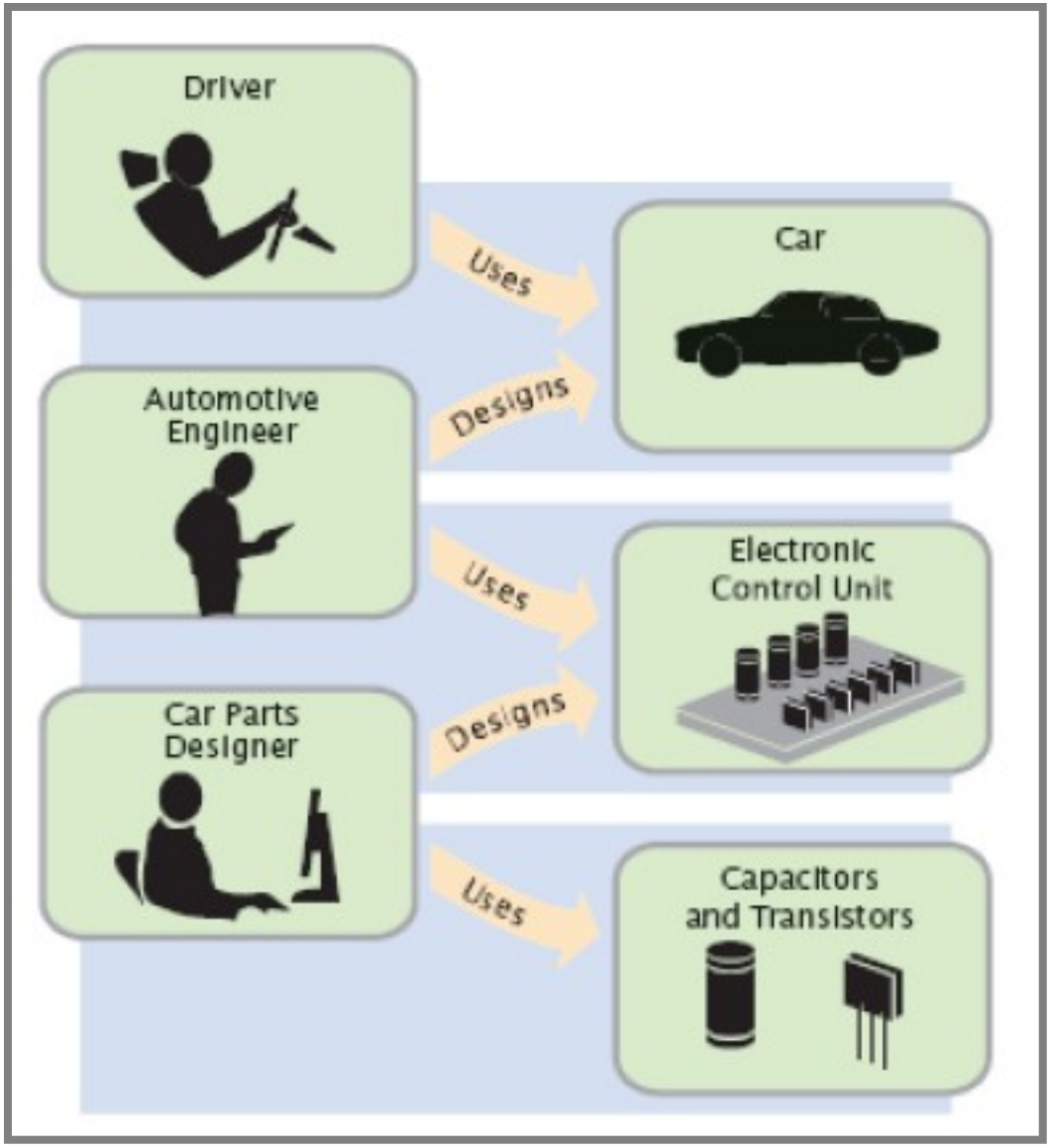
- Hides its details
- Encapsulation

Study sections 3.1 … 3.3, 3.6, and 3.7 from your text book.

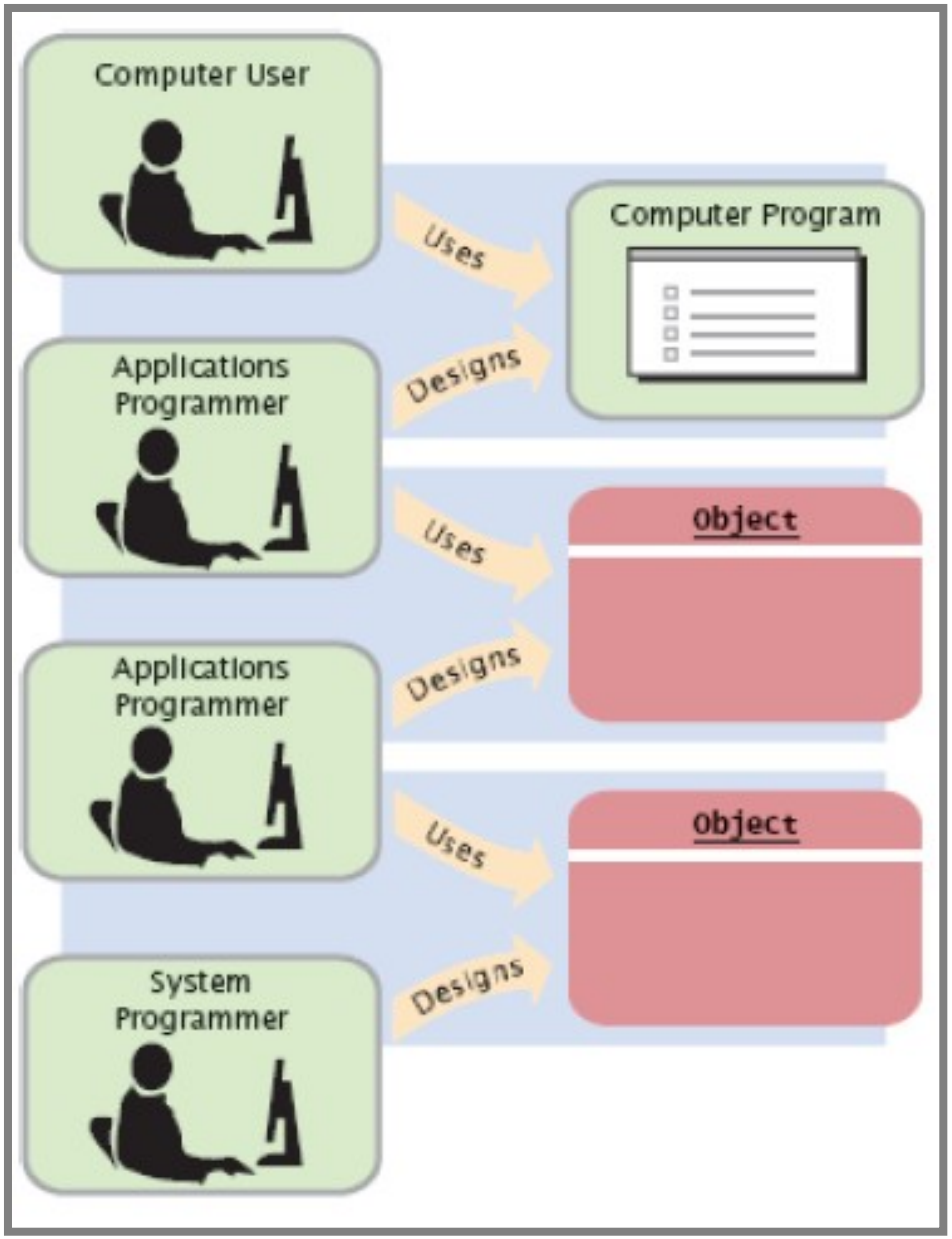
Levels of Abstraction: A Real- Life Example

- Users of a car do not need to understand how black boxes work
- Interaction of a black box with outside world is well-defined
- Encapsulation leads to efficiency:
  - Mechanic deals only with car components (e.g. electronic control module), not with sensors and transistors
  - Driver worries only about interaction with car (e.g. putting gas in the tank), not about motor or electronic control module

Levels of Abstraction:A Real-Life Example



Levels of Abstraction: Software Design



Levels of Abstraction: Software Design

Old times programmers 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:

Encapsulate routine computations to software black boxes

Levels of Abstraction: Software Design

- Abstraction used to invent higher-level data types.
- Structured Programming: Encapsulate routine computations (methods) to software black boxes.
- Object-Oriented Programming: objects are black boxes.
- Encapsulation: Programmer using an object knows about its behavior, but not about its internal structure.

Continued…

Levels of Abstraction: Software Design

- In software design, you can design good and bad abstractions with equal facility; understanding what makes good design is an important part of the education of a software engineer

1) First, define required operations.
2) Then develop use cases (how a program would carry out the operations).
3) Finally implement it.

Example

Let begin with a simple example: a tally counter

whenever operator clicks the button, the counter value increments by one.


```
Counter tally = new Counter();
tally.click();
tally.click();
int result = tally.getValue(); // set the result to 2
```

Instance & instance variables

An instance of a class is an object of the class.

In this example, tally is an instance of class Counter.

Each Counter object needs a variable that keeps track of the number of button clicks.

An object stores its data in instance variables.

Thus, an instance variable is a storage that is present in each object of the class

Instance variables

Syntax

```
public class ClassName
{
   private typeName variableName;
   . . .
}
```

```
public class Counter
{
   private int value;
   . . .
}
```

Each object of this class has a separate copy of this instance variable.

Instance variables should always be private.

Type of the variable

Declaration of an instance variables:

- An access specifier (private)
- Type of the instance variable (such as int)
- The name of the instance variable (such as *value*)

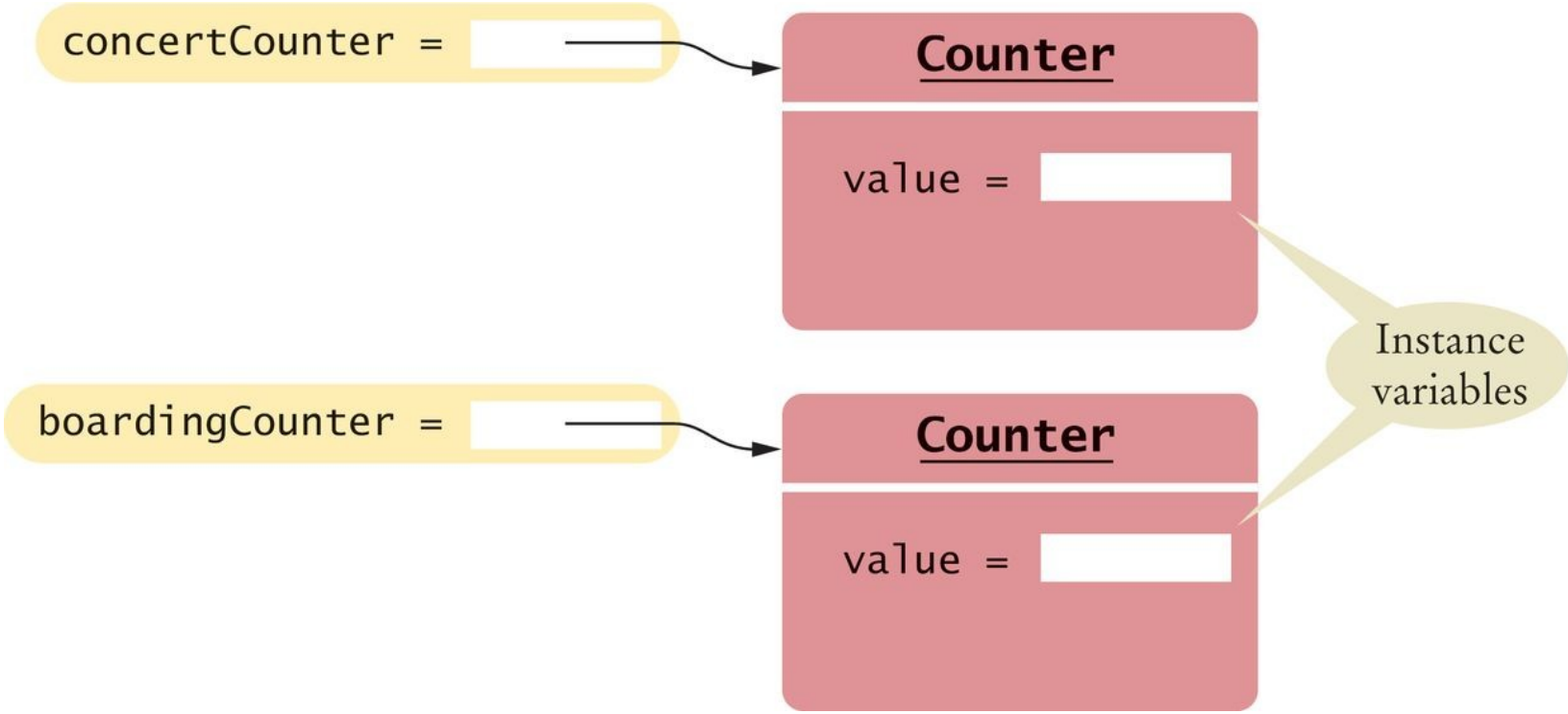
instance variables

Each object has its own instance variables.

Example:

Counter concertCounter = new Counter();

Counter bordingCounter = new Counter();



Methods

In this example, the click method advances of the counter value by 1.

```
public void click()
{
  value = value +1;
}
```

and the getValue() returns the current value.

```
public  int getValue()
{
  return value;
}
```

Example

Example: design BankAccount class

First define required operations:

- Deposit money
- Withdraw money
- Get balance

```
deposit
withdraw
getBalance
```

Designing the Public Interface of a Class: Methods

Develop use cases of the class (how class is going to be used).

```
harrysChecking.deposit(2000);
harrysChecking.withdraw(500);
double balance = harrysChecking.getBalance();
```

Define Methods

- Access specifier (such as **`public`**)
- Return type (such as `String` or `void`)
- Method name (such as `deposit`)
- List of parameters (`double amount` for `deposit`)

Example

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

Method Definition

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

Example:

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

Purpose:

To define the behavior of a method

Instance Fields

- An object stores its data in instance fields.
- The class declaration specifies the instance fields:

```
public class BankAccount
{
    private double balance;

    ...
}
```

Instance Fields

- An instance field declaration consists of the following parts:
  - access specifier (usually private)
  - type of variable (such as double)
  - name of variable (such as balance)
- Each object of a class has its own set of instance fields
- You should declare all instance fields as private

Accessing Instance Fields

- Method of an object can access the private instance field:

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

- private instance field cannot be accessed by outsiders. (Encapsulation)

Constructor Definition

- A constructor method initializes the instance variables.

Constructor's name must be the same as the class name.

```
public BankAccount()
{
   // body--filled in later
}
```

- Constructor body is executed when new object is created.
- Statements in constructor body will set the internal. data/state of the object that is being constructed.
- All constructors of a class have the same name.
- Compiler can tell constructors apart because they take different parameters.

Constructor Definition

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

Example:

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

Purpose:

To define the behavior of a constructor

BankAccount Public Interface

- The public constructors and public methods of a class form the public interface of the class.
- Develop declaration of the methods and the constructors of the class.

```
public class BankAccount
{
   private double balance;
   // Constructors
   public BankAccount()
   {
      // body--filled in later
   }
   public BankAccount(double initialBalance)
   {
      // body--filled in later
   }
```

BankAccount Public Interface (continue)

```
   // Methods
   public void deposit(double amount)

   {
      // body--filled in later
   }

   public void withdraw(double amount)
   {
      // body--filled in later
   }
   public double getBalance()
   {
      // body--filled in later
   }
   // private fields--filled in later
}
```

Commenting on the Public Interface

```
/**
   Withdraws money from the bank account.
   @param the amount to withdraw
*/
public void withdraw(double amount)
{
   // implementation filled in later
}
/**
   Gets the current balance of the bank account.
   @return the current balance
*/
public double getBalance()
{
   // implementation filled in later
}
```

Class Comment

```
/**
   A bank account has a balance that can
   be changed by deposits and withdrawals.
*/
public class BankAccount
{
   . . .
}
```

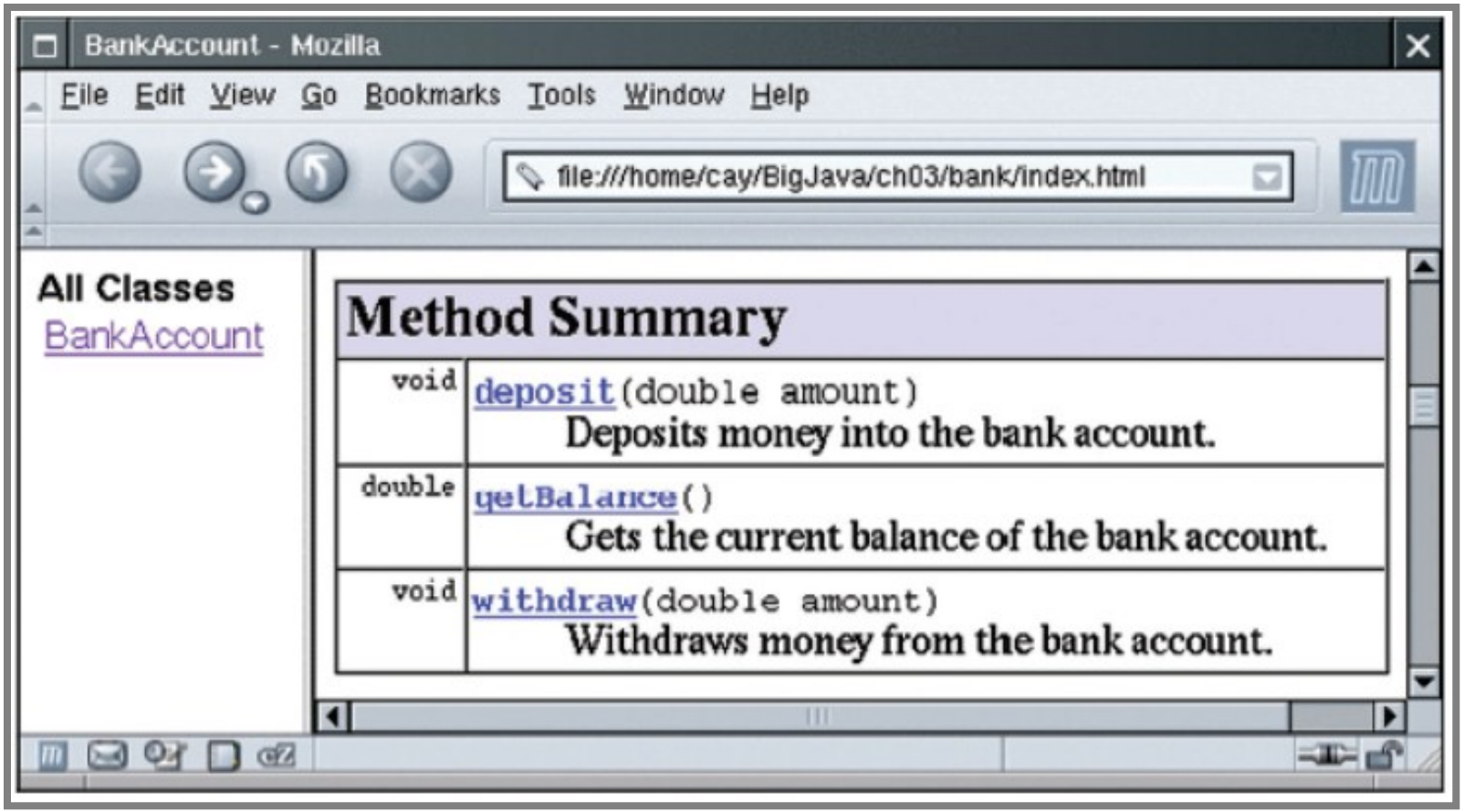
- Provide documentation comments for
  - every class
  - every method
  - every parameter
  - every return value.

Now type :

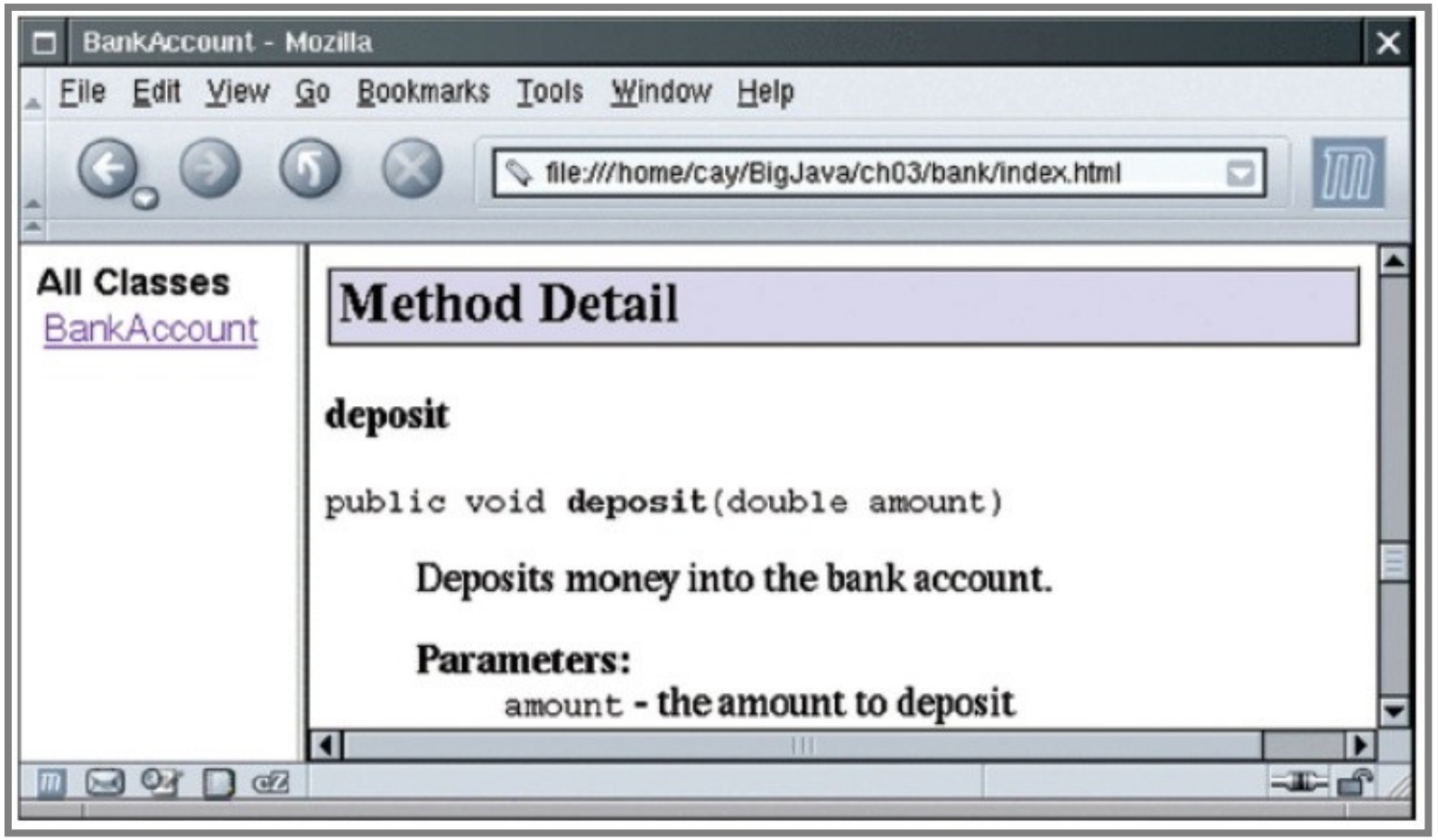
Javadoc BankAccout.java

that will create BankAccount.html

Javadoc Method Summary



Javadoc Method Detail



Implementing Constructors

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

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

Constructors do not have return type.

Example: 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;
          balance = newBalance;
31:     }
33:
34:     /**
35:        Withdraws money from the bank account.
36:        @param amount the amount to withdraw
37:    */
```

Continued…

File BankAccount.java

```
38:    public void withdraw(double amount)
39:    {
40:       balance = balance - amount;
41:    }
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

- Test Harness: a class with a main method that contains statements to test another class.
- Typically carries out the following steps:
  1. Construct one or more objects of the class that is being tested.
  2. Invoke methods of the objects.
  3. Print out one or more results.

Continued…

Testing a Class

- Details for building the program vary. In most environments, you need to carry out these steps:
  1. Make a new sub folder for your program
  2. Make two files, one for each class
  3. Compile both files
  4. Run the test program

File 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);
15:       System.out.println(harrysChecking.getBalance());
16:    }
17: }
```

Categories of Variables

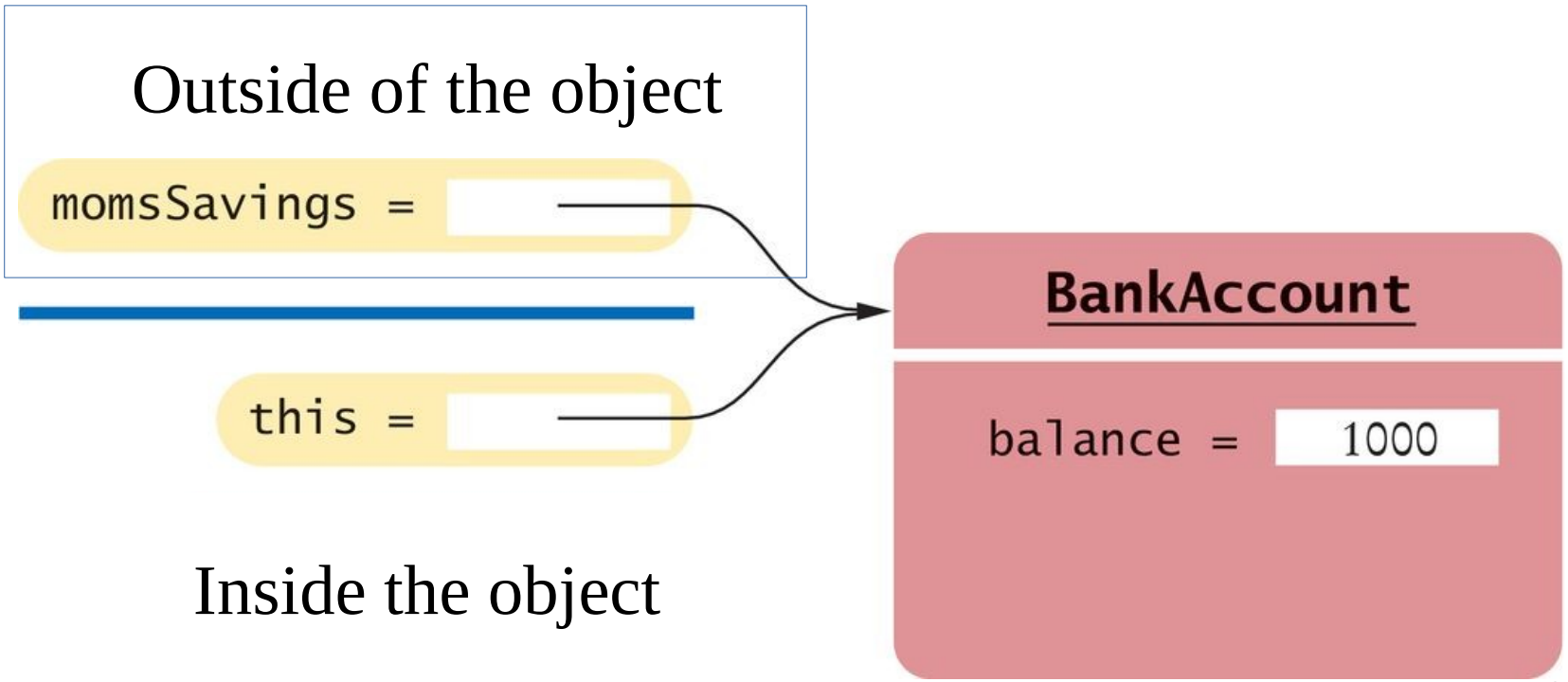
- Categories of variables
  - Instance fields (`balance` in `BankAccount`)
  - Local variables (newbalance in `deposit` method)
  - Parameter variables (`amount` in `deposit` method)
- An instance field belongs to an object.
- The fields stay alive until no method uses the object any longer

Local variables shadows an instance variable with the same name.

Implicit and Explicit Method Parameters

- The implicit parameter of a method is the object on which the method is invoked.
- The **`this`** reference denotes the implicit parameter.

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



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;
}
```

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

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 on Chapter 9).
- When you refer to an instance field in a method, the compiler automatically applies it to the **`this`** parameter.

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

Garbage Collector

- In Java, the *garbage collector* periodically reclaims objects when they are no longer used
- Local and parameter variables belong to a method
- Instance fields are initialized to a default value, but you must initialize local variables

Summary

Each object of a class has its own set of instance variables.

Private instance variables can only be accessed by methods of the same class.

Encapsulation is the process of hiding implementation details.

Constructors name is always the same as the class name.

Constructors set the initial data for objects.

Instance variables are initialized to a default value, but you must initialize the local variables.

Summary (continue)

The this reference denotes the implicit parameter.

Local variables shadows an instance variable with the same name.

A method call without an implicit parameter is applied to the same object.