Computer Science 2

Lecture 1: Part 1

Objects and Classes

Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Run
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes

Problem

We have to write a program for a bank. The program has to allow a user to **deposit** and **withdraw** money from her bank account.

Problem Analysis

When we analyze a problem domain we identify:

- **Objects;**
- **Possible states of the objects; and**
- **Relationships between the objects.**

In addition, we identify classes. A *class* is a set of objects in the problem domain that are unified by a *reason*. A *reason* may be a similar appearance, structure, or function.

Example. The set: ***{children, photos, cat, diplomas}*** can be viewed as a class "Most important things to take out of your apartment when it catches fire".

Problem Analysis

When we are handed a problem the first thing to do is to figure out what is going on – to analyze its domain

When we analyze a problem domain we identify:

- **Objects and classes (things), and their properties**
- **What objects can do and what can be done to them**
- **Relationships between the objects**

Put simply, we are looking for:

- **Nouns**
- **Verbs**
- **Relationships**

Problem Analysis

For now we will stick with just the first two

Our goal is to end up with a computer program that simulates the important features of the system under analysis

This is the basic idea behind Object Oriented Programming

To solve a problem, simulate the system the problem is a part of, then run the simulation with various parameters

Problem Analysis

For our problem domain we may find:

- **Bank-account objects with balances**
- **One can deposit, withdraw, and ask for the balance of an account**

We assume that the bank accounts are not related.
Thus, since we don't have other classes of objects,
no object relationships appear in our analysis.

Balance

Bank Account

Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Run
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes

Objects vs Classes

- People naturally notice that, although they only ever see individual things, these things seem to belong together in natural “types” or ”kinds” of things
  - You only ever see individual bicycles
  - But you have an idea of “bicycle” that encompasses all of them (and more)
  - Think Plato and Forms
  - How this happens is a bit of a mystery
- This abstract idea of “bicycle” is useful

Objects vs Classes

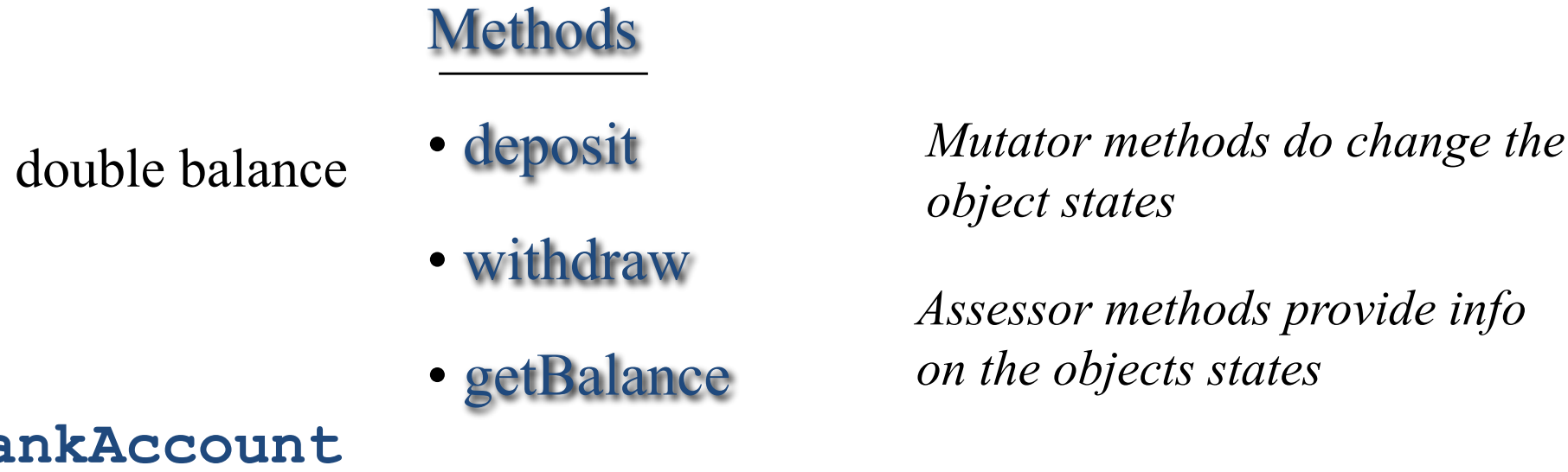
- For example, if someone says they bought a new bicycle, even without seeing it you know some things about it
  - It has two wheels, pedals, a seat, etc.
  - It can be ridden places, it can start and stop, etc.
- In programming-speak
  - An object is a particular bicycle
  - A class is the idea of “bicycle”

Objects in OO Languages

- Objects identified during problem analysis are represented as ***Objects*** in OO languages;
- An ***object*** in OO languages consists of ***data*** encapsulated with a set of ***methods*** which operate only on these data;
- The ***data*** of an object determine the ***state*** of the object;
- The ***methods*** associated with an object can ***change the state*** of the object, or provide info on ***the state*** of the object, or determine relationships of the object with other objects.

Example: Objects in OO Languages

- In our case a **`BankAccount`** object consists of a double variable Balance:
  - *A variable is an item of information in memory whose location is identified by a symbolic name.*
- The object is associated with three methods: deposit, withdraw and getBalance.



Classes in OO Languages

- Of course, given what we know about banks, the bank will want to have multiple bank accounts on deposit
- These accounts will all be similar
  - Each will have a balance, the ability to deposit and withdraw money, and so forth
- Although they will belong to different people
- To do this you need a class that represents the idea of a bank account
- Then you can make many different bank accounts using the class as a template

Classes in OO Languages

- Classes identified during problem analysis are represented as ***Classes*** in OO languages;
- A ***class*** in OO languages is the prototype for the objects it represents.
- The **structure of a class** is determined by:
  - the **fields** which represent the state of an object of that class;
  - the **methods** associated with the objects the class represents.
- This structure will be shared by all objects that belong to this class

Example: Classes in OO Languages

```
public class BankAccount
{  public BankAccount(double initialBalance)
   {  balance = initialBalance;
   }
   public void deposit(double amount)
   {  double newBalance = balance + amount;
      balance = newBalance;
   }
   public void withdraw(double amount)
   {  double newBalance = balance - amount;
      balance = newBalance;
   }
   public double getBalance()
   {  return balance;
   }
   private double balance;
}
```

Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Run
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes

Instance Fields

Instance fields are variables that store the states of the objects.

An instance field declaration consists of:

· access specifier (such as **`private`** or **`public`**);

· type of variable (such as **`double`**);

· name of variable (such as **`balance`**).

```
<access specifier> <variable type> <variable name>;
```

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

Instance Methods

The methods change the object state or compute the relationships with other objects. A method definition in a class consists of:

- access specifier (such as **`public`**);
- return type (such as **`double`** or **`void`**);
- method name (such as **`withdraw`**);
- list of parameter variables (empty for **getBalance()**);
- method body in { }.

```
<accessSpecifier> <returnType> <methodName> (<Type> <Name>,...)
{
 <method body>
}
```

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

The definitions of all the methods of a class is the application public interface of the class!

The Return Statement

- The return statement returns a value of an expression and exits the method immediately.
- If the type of the method is different from **`void`** then the returned value has to be of the same type as the type of the method.

`return <expression>;`

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

The Return Statement

- If the type of the method is **`void`** then there is no returned value: **`return`**;

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

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

Example: Methods

```
public class BankAccount
{   ……
   public void deposit(double amount)
   {   double newBalance = balance + amount;
       balance = newBalance;
   }
   public void withdraw(double amount)
   {   double newBalance = balance - amount;
       balance = newBalance;
   }
   public double getBalance()
   {   return balance;
   }
   private double balance;
}
```

Variable Access

- Every method has access to three different sorts of variables:
- Instance fields
  - Which we have seen, and will see again
- Parameter variables
  - Or just parameters
- Local variables
- We will discuss the second two presently

Parameter and Local Variables

- Parameter variables are the variables in the method's definition. When a method is called with actual values, their types have to match the types of the parameter variables!
- Local variables are the variables defined in the method's bodies.

```
public class BankAccount
{   ……….
  public void withdraw(double amount)
  {  double newBalance = balance - amount;
     balance = newBalance;
  }
  …….
}
```

Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Run
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes

Encapsulation

How an object performs its duties is hidden from the outside world.

- Methods can be used without the knowledge of the inner workings
- Inner workings can be modified without impacting use (as long as the interface is unchanged)

This idea is called the *separation of implementation and interface*

Encapsulation

- The purpose of encapsulation is to make it easier on anybody who wants to use the code
- The idea is that it is easier to use code if you know what it will do, but do not have to worry about how

Example

```
public class BankAccount
{ public BankAccount(double nBalance)
 {  balance = nBalance;
 }
 public void deposit(double amount)
 {  double newBalance = balance +
amount;
    balance = newBalance;
 }
 public void withdraw(double amount)
 {  double newBalance = balance - amount;
    balance = newBalance;
 }
 public double getBalance()
 {  return balance;
 }
 private double balance;
}
```

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

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

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

Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Run
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes

Constructors

- A class constructor is a special method that creates an object of the class and initializes instance variables.
- A constructor has the same name as the class.
- Unlike other methods constructors don't have return type.

```
<access specifier> <class name> (<Type> <Name>,...)
{<constructor's body>} ;
```

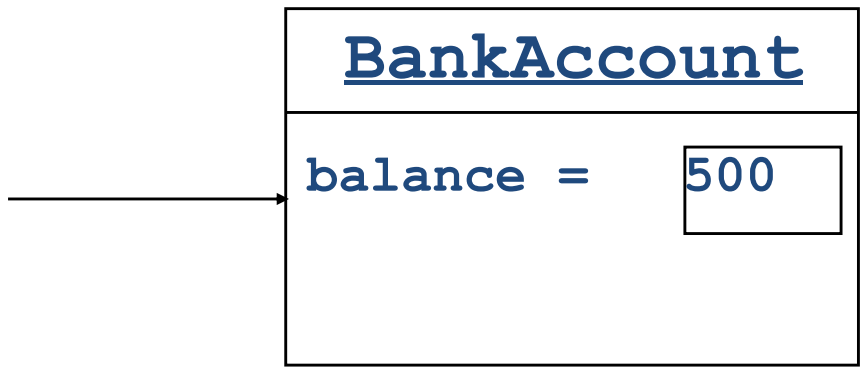
```
public class BankAccount
{  public BankAccount(double initialBalance)
   {  balance = initialBalance;
   }
……….
    private double balance;
}
```

Operator `new`

- The **`new`** operator creates an object of a class using the class constructor and returns the object reference.
- We create an object with **`new`** as follows:
  - We type the **`new`** keyword;
  - We give the name of the class;
  - We supply construction parameters (if any).

Example.

```
new BankAccount(500);
```



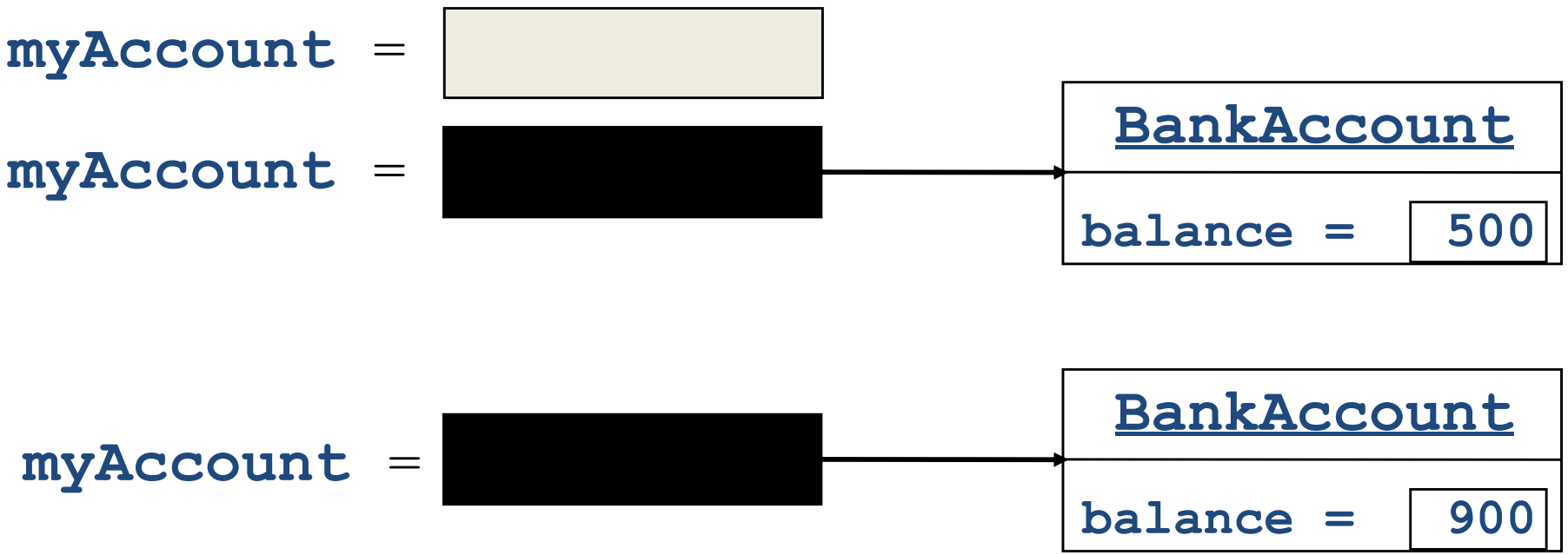
Reference Variables

- To manipulate the object, its reference has to be stored in a reference variable.
- The type of the reference variable has to match the type of the object created.
- To be able to use an object we execute 4 steps:
  - Define a reference variable; (doesn't create the object)
  - Construct an object with the **new** operator;
  - Store the object location in a reference variable;
  - Call methods on the object variable.

```
BankAccount myAccount;

myAccount = new BankAccount(500);

myAccount.deposit(500000);
```

Reference Variables and Object Manipulation

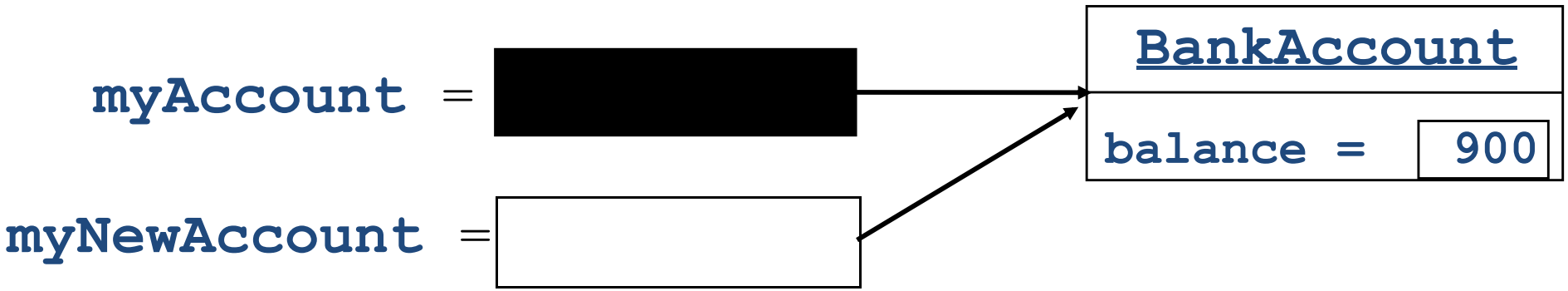
```
BankAccount myAccount;

myAccount = new BankAccount(500);

myAccount.deposit(400);
```



Multiple Object Variables

Multiple object references can refer to the same object.

```
BankAccount myAccount  = new BankAccount(900);

BankAccount myNewAccount = myAccount;
```



Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Run
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes

Class Run: Option 1

- Provide a main method in the class that is being run. The main method has:
  - to construct one or more objects of the class;
  - to invoke one or more methods;
  - to print one or more results.

```
public class BankAccount
{   ……….
  public static void main(String[] Args)
  {   BankAccount b = new BankAccount(0);
   b.deposit(100);
   b.withdraw(50);
   System.out.println(b.getBalance());
  }   …….
}
```

Class Run: Option 2

- Create a separate class with the same `main` method.
- To run, combine the class to be run and the separate class:
  - Make a new subfolder;
  - Make two files, one for each class;
  - Compile both files;
  - Start the running class.

```
public class BankAccountRun
{  public static void main(String[] Args)
   {   BankAccount b = new BankAccount(0);
    b.deposit(100);
    b.withdraw(50);
    System.out.println(b.getBalance());
   }
}
```

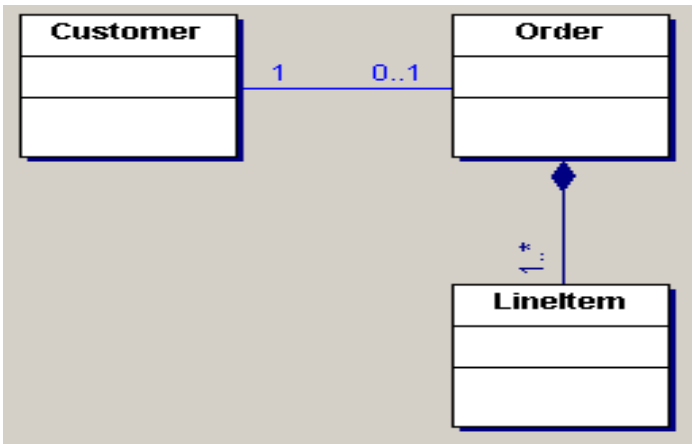
Class Run: Option 2

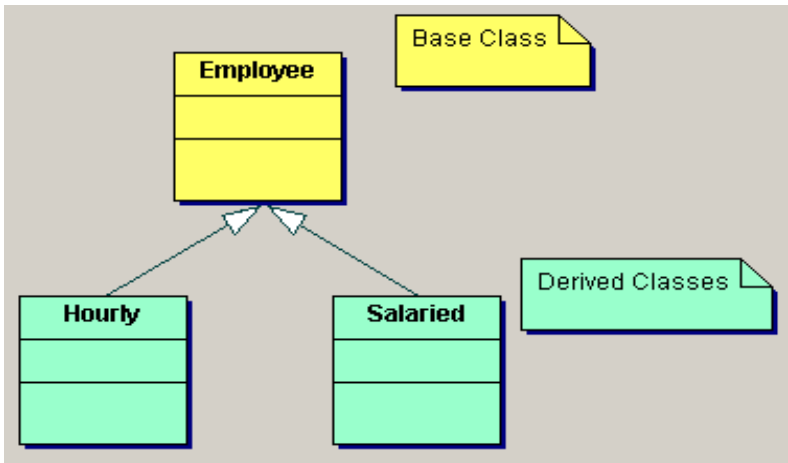
- Option 1 is easier for small programs that only use one class
- Option 2 is better for larger programs, when multiple classes are used

Object-Oriented Programming

Now that we have the test program, a program that uses objects, we can define Object-oriented programming.

Object-oriented programming is a discipline of programming where each program is a simulation of the domain of interest. The program is populated by objects, and these objects communicate with one another to solve a problem using methods.





Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Test
- Constructors, Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes

More about Constructors

- A class can have no constructors, one constructor, or more than one constructor.
- A constructor is supposed to initialize all the instance fields. If not, then these variables receive their default values:
  - Numerical fields receive value **0 (0.0);**
  - Reference fields receive value **`null`**.

```
public class BankAccount
{  public BankAccount()
   {  balance = 0.0;}
   public BankAccount(double initialBalance)
   {  balance = initialBalance;}
   ……..
   private double balance;
}
```

Instance Variable Lifetime

- Instance fields are created when an object is constructed. They 'die' when the object 'dies' (no object variable refers the object).

```
public class BankAccountTest
{ public static void main(String[] Args)
  {   BankAccount b = new BankAccount();
   b.deposit(100);
      b = null;
   .....
  }
}
```

Overloading

- Overloaded methods are methods with the same name but different
  - Numbers of parameters, or
  - Parameter types
- The simplest form of overloading is seen with constructors
- Let us look at how the BankAccount class might work with overloaded constructors

Overloading

- If we call **`new BankAccount()`**, then the first constructor is called;
- If we call **`new BankAccount(20)`**, then the second constructor is called;
- If we call **`new BankAccount("abba")`**, then the compiler generates an error.

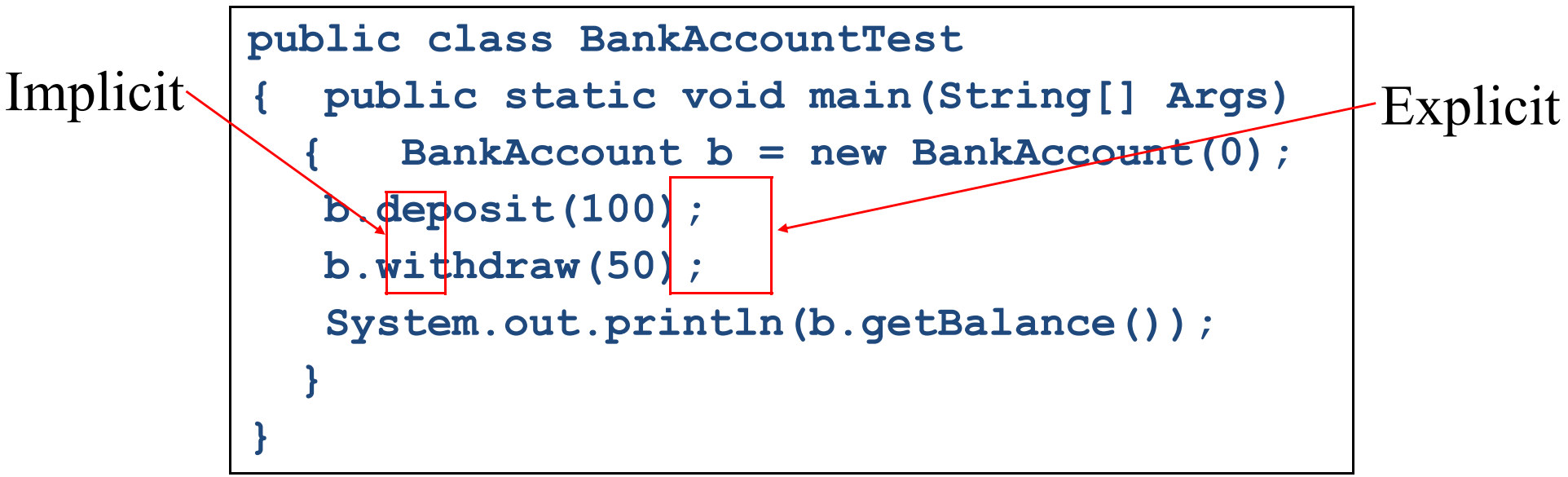
```
public class BankAccount
{  public BankAccount()
   {  balance = 0.0;}
   public BankAccount(double initialBalance)
   {  balance = initialBalance;}
   ……..
   private double balance;
}
```

Explicit and Implicit Parameters

- A parameter is explicit if it is explicitly named in the method definition. Otherwise, it is called an implicit parameter.
  - Generally, the implicit parameter refers to the object upon which the method is being called
- In Java this does not matter much
  - In Python this is a big deal

Explicit and Implicit Parameters

- We can see what the terms mean with a simple example



Parameters: Initialization and Life Time

- Parameter variables are initialized with the values supplied in the method call. Thus, the call has to be correct. Otherwise, the compiler generates an error.
- Parameter variables are created when methods start. They 'die' when the methods exit.

```
public class BankAccountTest
{  public static void main(String[] Args)
   {   BankAccount b = new BankAccount(0);
    b.deposit(100);
    b.withdraw();
    System.out.println(b.getBalance());
   }
}
```

The compiler generates an error here!

Local Variables: Initialization and Life Time

- Local variables are initialized in methods. Otherwise, the compiler generates an error.
- Parameter variables are created when method execute their definition. They 'die' when the methods exit.

```
public class BankAccount
{   ……
  public void deposit(double amount)
  {  double newBalance;
     balance = newBalance;
  }
  ……
  private double balance;
}
```

The compiler generates an error here!

Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Test
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes

Commenting the Public Interface

- Part of the point of encapsulation is that other people should not have to read your code to find out what it does or how to use it
- Then how do they find out?
- They read the documentation you wrote
- Java makes creating documentation for your code easier by supporting a form of commenting called Javadoc
  - Named after the program that reads your comments and creates the documentation

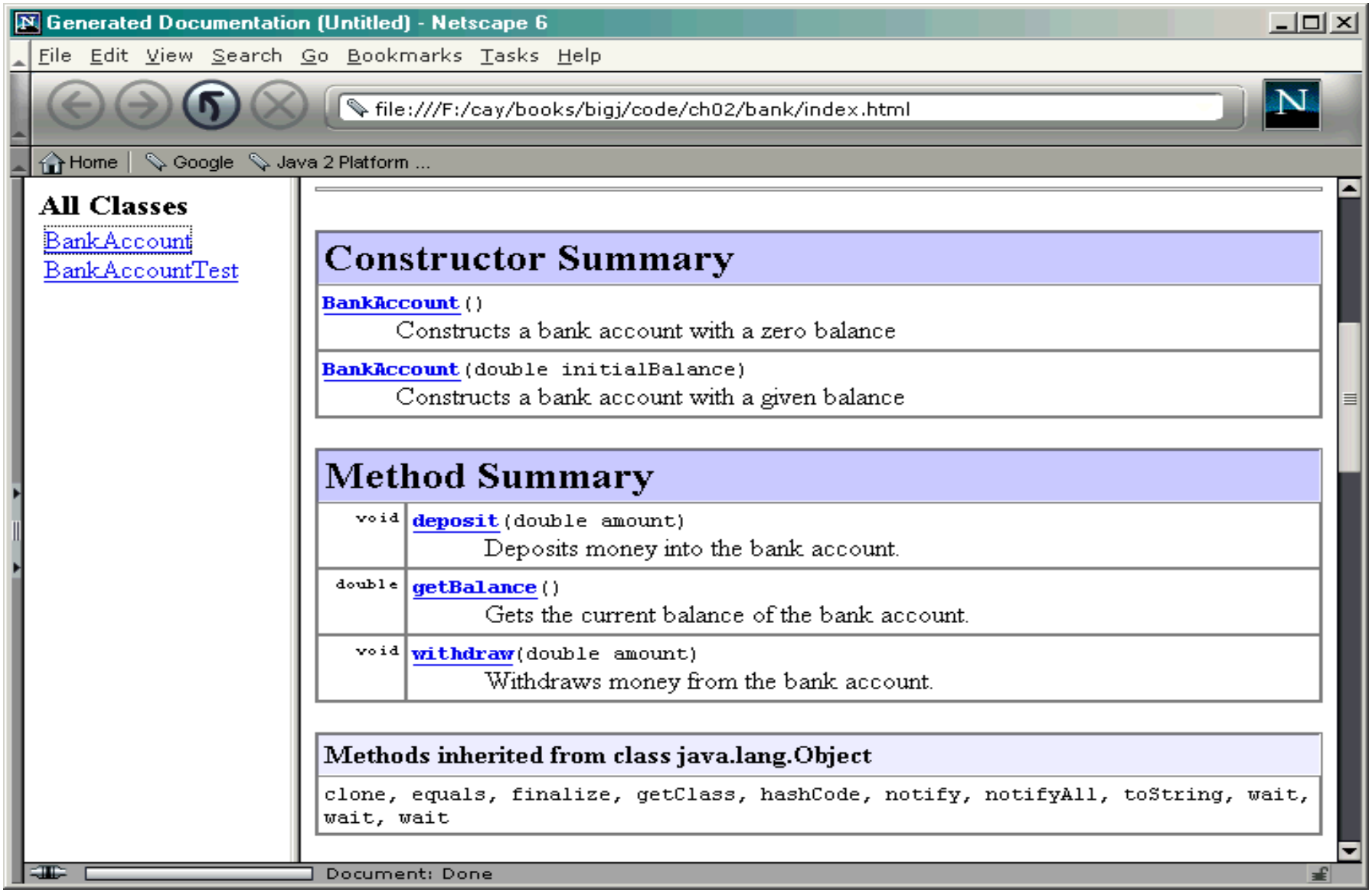
Commenting the Public Interface

- A Javadoc comment starts with `/**` and end with `*/` .
- For each method parameter you supply a line that starts with `@param` tag followed by the parameter name and description.
- For each non-void method you supply a line that starts with `@return` tag followed by the parameter name and description.
- When ready, start `javadoc` on your file.

Commenting the Public Interface

```
/**
A bank account has a balance that can be changed
by deposits and withdrawals.
*/
public class BankAccount
{ ……..
 /**
 Constructs a bank account with a given balance
 @param initialBalance the initial balance
 */
  public BankAccount(double initialBalance)
  {  balance = initialBalance;
  }
   …..
  private double balance;
}
```

Commenting the Public Interface



Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Test
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes

Importing Classes from Packages

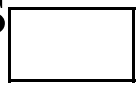
- Java classes are grouped in packages. If you use a class from another package (other than the java.lang package), you have to import the class at the beginning of your program.

```
import <packageName>.<ClassName>;
```

- Example:
  - `import java.awt.Rectangle;`
  - `import java.awt.*;` **(import all the classes)**

Overview

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Test
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes



Designing and Implementing Classes

- Designing a good class can be difficult
- There are some steps that will help

1. What does the class need to do?
2. What will the method names be?

- Document the public methods
- What private variables and methods are needed?
- What constructors are needed?
- Implement the methods
- Test, test, test

Designing and Implementing Classes

- **Find out what you are asked to do with an object of the class.**

Suppose you implement a `Person` class. The application to be written requires:

- To get the name of the person;
- To set a new name of the person .

Then make sure your class has methods that do this

Designing and Implementing Classes

- Generally, classes need four kinds of methods
- Creators
  - Constructors
- Readers
  - Methods that return some information about the state of an object
- Updaters
  - Methods that allow the state of the object to be changed in some way
- Destructors (or Deleters)
  - Methods that should be called when an object is no longer needed
  - For example, if an object has a connection to a database, it should have a method that allows the user to end that connection before the object if thrown away

Designing and Implementing Classes

- Collectively these types of methods are known as CRUD
  - As in, your class should be CRUD-dy
- Readers often have names that start with get
  - getBalance
  - Also known as getters
- Updaters often have names that start with set
  - setBalance
  - Also known as setters
- Although these are not written in stone
  - The deposit method is a good example of a setter that does not start with “set”
- The point is to name things so that a potential user can understand what the method does

Designing and Implementing Classes

- **Find names for the methods**

  Come up with the method names and apply them to a sample object, like this:

```
Person suspect = new Person("John Lee");
suspect.getName();
suspect.setName("John Smeet");
```

Designing and Implementing Classes

- Document the public interface

```
public class Person
{ /**
      Set new name of the person
      @param name new name of the person
   */
    public void setName(String newName)
    {}
   /**
      Get the name of the person
      @return the name of the person
   */
    public String getName()
    {}
}
```

Designing and Implementing Classes

- Determine instance variables

```
private String name;
```

Designing and Implementing Classes

- Determine constructors

```
/**
   Constructs a new person object
   @param name name of the person
*/
 public Person(String newName)
 {
     name = newName;
 }
```

Designing and Implementing Classes

- Implement methods; i.e., the class is ready.

```
/**
   Set new name of the person
   @param name new name of the person
*/
 public void setName(String newName)
 {
     name = newName;
 }
/**
   Get the name of the person
   @return the name of the person
*/
 public String getName()
 {
     return name;
 }
```

Designing and Implementing Classes

- Test your class

```
public class BankAccountTest
{
   public static void main(String[] Args)
   {
       Person suspect = new Person("John Lee");
       System.out.println(suspect.getName());
       suspect.setName("John Smeet");
       System.out.println(suspect.getName());
   }
}
```

Concepts Covered in the Lecture

- Problem Analysis
- Objects and Classes
- Class Elements: Instance Fields, and Methods
- Encapsulation Principle
- Object Creation
- Class Test
- Overloading Methods, Variable Initialization and Lifetime
- Commenting Public Interface
- Class Import
- Approach to Design and Implement Classes