Principles of Computer Science I

Prof. Nadeem Abdul Hamid
CSC 120 – Fall 2006
Lecture Unit 3 - Implementing Classes

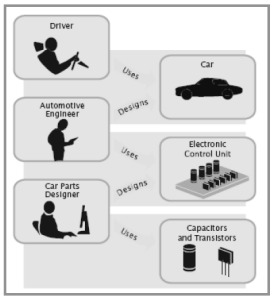
Lecture Outline

- Implementing classes, methods, constructors
- Instance fields and local variables
- Documenting code
  - *Javadoc*

Black Boxes

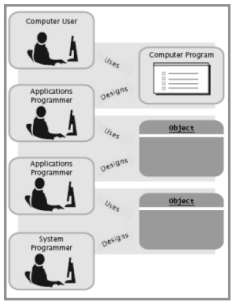
- 'Black box' - device whose inner workings are hidden
  - Car - electronic control module
  - Java - objects
- *Encapsulation* - hiding unimportant details
- *Abstraction* - taking away inessential features until essence of concept remains

Levels of Abstraction: Car



- Users do not need to understand the 'black boxes'
  - Leads to efficiency, ease-of-use
- Interaction of black box with outside world is well-defined
  - Drivers interact using pedals, buttons, etc.
  - Mechanic tests engine control module (ECM) sends the right firing signals to the spark plugs
  - ECM manufacturers use transistors and capacitors, black boxes magically produced by an electronics component manufacturer

Levels of Abstraction: Software



- Old times: computer programs manipulated primitive types such as numbers and characters
  - Too much for human programmers
  - Solution: Design software 'black boxes'
- Abstraction: invent higher-level data structures
- Encapsulation: programmer using object knows behavior, not internal implementation

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
- First, define behavior of a class; then, implement it

Designing a Class: BankAcct

- Behavior of a bank account
  - Deposit money
  - Withdraw money
  - Get balance
- Method definitions
  - Access specifier
  - Return type
  - Name
  - Parameter list
  - Body

Syntax: Method Definition

```
accessSpecifier returnType
                        methodName(paramType paramName, …)
{
   method body
};
```

Example:

```
public void deposit( double amount ) {
   …
}; // end method deposit
```

Purpose:
To define the behavior of a method

Constructors

- A constructor initializes the internal data of an object
  - Is a special method
  - Constructor name must be the same as the class
- Constructor body is executed when a new object is *instantiated*
- All constructors of a class have the same name
- Compiler can tell constructors apart because they take different parameters

Syntax: Constructor Definition

```
accessSpecifier ClassName(paramType paramName, …)
{
   constructor body
};
```

Example:

```
public BankAccount( double initialBalance ) {
   …
}; // end constructor
```

Purpose:
To define the behavior of a constructor

BankAccount Public Interface

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

```
public class BankAccount {

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

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

    // 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
}
```


Syntax: Class Definition

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

Example:

(see previous slide)

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

Using BankAccount

- Write code to instantiate (create) two accounts with some initial balances, then transfer money from one account to another
- Write code to empty (withdraw all money from) a bank account

Comments

- Ignored by the computer (compiler)
- Comments make programs easier to understand for humans
- Use comments liberally, but make them meaningful
- Two forms of Java comments
  - Comments between `/*` and `*/` can extend over several lines
  - Using two slashes `//` makes the rest of the line become a comment

javadoc Commenting Style

- Standard form for documentation comments
- javadoc automatically generates HTML (web) pages describing your classes based on comments in source code
- javadoc comment starts with `/**`
  - First line describes method/class purpose
  - For each parameter, give line starting with `@param`
  - Supply line starting with `@return` describing return value

javadoc Method Comments

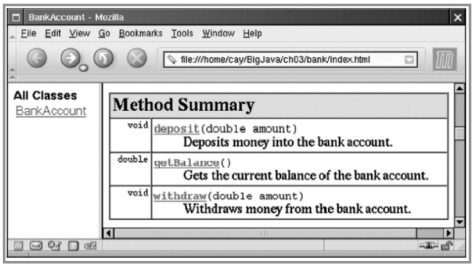
```
/**
   Withdraws money from the bank account.
   @param amount the amount to withdraw
*/
public void withdraw(double amount)
{
   double newBalance = balance - amount;
   balance = newBalance;
}

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


javadoc Class Comment

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

- Provide comments for
  - Every class
  - Every method
  - Every parameter
  - Every return value

javadoc Output



Instance Fields

- Object stores its data in *instance fields*
- Field: storage location inside memory
- Instance: an object of a class

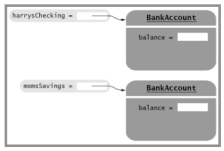
```
public class BankAccount
{
   . . .
   private double balance;
}
```

- Instance field declaration:
  - Access specifier (usually `private`)
  - Type of the field (like `double`)
  - Name of the field (like `balance`)

Instance Fields (Syntax)

- Every object of a class has its own set of instance fields

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



Example:
(see previous slide)

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

Accessing Instance Fields

- Methods of the same class can access `private` fields
- Methods/code outside the class cannot
- Encapsulation = Hiding data (fields) and providing access through public interface (methods)

Implementing Constructors and Methods

- Constructors contain code to initialize instance fields of object
- Some methods do not return a value
- Other methods return a result (`getBalance`)
  - Use a **`return`** statement to exit a method immediately/return a value
- *BankAccount.java*

```
public class BankAccount {

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

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

    // 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 double balance;
}
```


Syntax: `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.

Constructor Call Example

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

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

Method Call Example

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

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

Checkpoint

- How would you implement the **`translate`** method of the **`Rectangle`** class?

Testing a Class

- Test class (sometimes called a 'driver class')
  - Class with a `main` method that contains code to test another class
- Typical steps:
  - Construct one or more objects of the class that is being tested
  - Invoke one or more methods
  - Print out one or more results
- Running test program (typical steps):
  - Make a new subfolder for your program
  - Make two files, one for each class
  - Compile both files
  - Run the test program

BankAccountTester.java

Summary: Designing and Implementing Classes

- Find out what an object of the class is supposed to do
- Specify the public interface
- Document the public interface
- Determine instance fields
- Implement constructors and methods
- Test the class

- Example: *Cash Register*

CashRegister.java
CashRegisterTester.java

Categories of Variables

- Three categories of variables
  - Instance fields (`balance` in `BankAccount`)
  - Local variables (`newBalance` in `deposit` method)
  - Parameter variables (`amount` in `deposit` method)

- Two important differences
  - Lifetime
  - Initialization

Variable Lifetimes

- Instance variables belong to object
  - Remain 'alive' until object is no longer being used
  - Java runtime system (virtual machine-JVM) contains program called *garbage collector* that periodically reclaims memory space of unused objects

- Local and parameter variables belong to a method
  - The 'die' when the method is exited

Variable Initialization

- Local variables must be initialized
  - Compiler will complain if you don't
- Parameter variables are initialized with argument values in the method call
- Instance fields are initialized with default value (either 0 or null)
  - Common cause of errors: forgetting to initialize instance variables in a constructor

31

Implicit Parameters

- The implicit parameter of a method is the object on which the method is invoked
- The **`this`** keyword refers to the object that is passed as the implicit parameter
- Every method has one implicit parameter
  - Using the name of an instance field inside the method means the instance field of the implicit parameter object
    - Can always use the keyword `this` inside a method to explicitly refer to the implicit parameter
  - Exception: static methods do not have implicit parameter (Ch. 9)

32

Calling One Constructor from Another

- Also uses the **`this`** keyword followed by parentheses as shorthand

33

Voting Machines