CHAPTER 3

IMPLEMENTING CLASSES

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)
- (To implement classes for drawing graphical shapes)

Levels of abstraction: A car 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
  - *Drivers interact with car using pedals, buttons, etc.*
  - *Mechanic can test that engine control module sends the right firing signals to the spark plugs*
  - *For engine control module manufacturers, transistors and capacitors are black boxes produced by a manufacturer*
- 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: Software design

- 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: Encapsulate routine computations to software black boxes
- Abstraction used to invent higher-level data types
- In object-oriented programming, objects are black boxes
- Encapsulation: Programmer using an object knows about its behavior, but not about its internal structure

Software design (cont.)

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

Specifying the public interface of a class

Behavior of bank account (abstraction):

- – deposit money
- – withdraw money
- – get balance

Specifying the public interface of a class: Methods

Methods of `BankAccount` class:

- `deposit`
- `withdraw`
- `getBalance`

We want to support method calls such as the following:

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

Specifying the public interface of a class: Method definition

Access specifier (such as `public`)

- return type (such as `String` or `void`)
- method name (such as `deposit`)
- list of parameters (`double amount` for `deposit`)
- method body in { }

Examples:

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

Syntax 3.1: Method definition

accessSpecifier returnType methodName(*parameterType parameterName*, . . .)

{

method body

}

Example:

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

Purpose:

To define the behavior of a method.

Specifying the public interface of a class: Constructor definition

- A constructor initializes the instance fields
- Constructor name = 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 of the object that is being constructed
- All constructors of a class have the same name
- Compiler can tell constructors apart because of different parameters

Syntax 3.2: Constructor definition

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

Example:

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

Purpose:

To define the behavior of a constructor.

Public interface of BankAccount class

The public constructors and methods of a class form its *public interface*.

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

BankAccount class (cont.)

```
   // 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 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.

Instance fields

- An object stores its data in instance fields
- Field: a technical term for a storage location inside a block of memory
- Instance of a class: an object of the class
- The class declaration specifies the instance fields public class

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

Instance fields

- An instance field declaration consists of the following:
  - *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

Instance fields

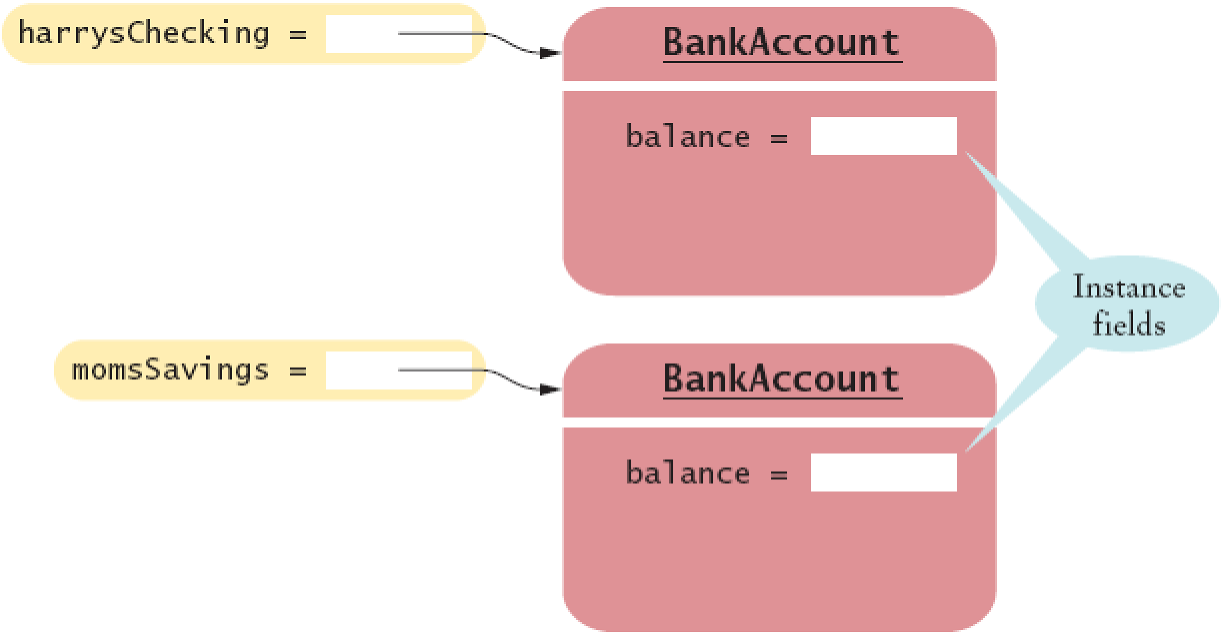

Figure 5 Instance Fields

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

- The deposit method of the BankAccount class can access the private instance field:

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

Accessing instance fields (cont.)

- Other methods cannot:

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

- *Encapsulation* is the process of hiding object data and providing methods for data access
- To encapsulate data, declare instance fields as private and define public methods that access the fields

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);`
  - *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*

Implementing methods

- Some methods do not return a value

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

- Some methods return an output value

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

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*

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.

The return value becomes the value of the method call expression.

The BankAccount class

```
public class BankAccount
{
  public BankAccount()
  {
    balance = 0;
  }

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

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

The BankAccount class (cont.)

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

  public double getBalance()
  {
    return balance;
  }

  private double balance;
}
```

Unit Testing

- *Unit test*: verifies that a class works correctly in isolation, outside a complete program.
- To test a class, write a tester class.
- *Test class*: a class with a main method that contains statements to test another class.
- Typically carries out the following steps:
  - *Construct one or more objects of the class that is being tested*
  - *Invoke one or more methods*
  - *Print out one or more results*

Unit Testing (cont.)

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

The BankAccount test class

```java
public class BankAccountTester
{
  public static void main(String[] args)
  {
    BankAccount harrysChecking = new BankAccount();
    harrysChecking.deposit(2000);
    harrysChecking.withdraw(500);
    System.out.println(harrysChecking.getBalance());
    System.out.println("Expected: 1500");
  }
}
```

Categories of variables

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

Categories of variables (cont.)

- An instance field belongs to an object
- The fields stay alive until no method uses the object anymore
- 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