Chapter 3 – Implementing Classes
Introduction

• In the previous chapter, we saw how to use objects
  – Declare an object
    • `Book aBook;`
  – Create an object
    • `aBook = new Book("Beloved","Toni Morrison");`
  – Call methods on an object
    • `aBook.getAuthor();`
Introduction

• Remember our goal is to write programs that accomplish something useful
  – How do objects fit in?
    • Encapsulation
    • Efficiency

  – We can also make our own types of objects…
Example Program

```java
public class Example
{
    public static void main(String args[])
    {
        Path p = new Path(3, 4);
        System.out.println("The distance is " + p.calcDistance()
            + " miles and the "
            + " angle is "
            + p.calcAngle());
    }
}
```
Example Program

• This is great, except... there is no `Path` class in the Java API

• The program would be easy if we had that type of object...

• So we’ll just write a `Path` class ourselves
Black Boxes

• What is a black box?

• Why do we call it a black box?

• What are some instances of a black box?
Black Boxes

• A black box works magically – we press a button and something happens

• We don’t know how it actually works

• *Encapsulation*: the hiding of unimportant details
Black Boxes

• Not everything is hidden in a black box
  – We can input some numbers
  – There is some sort of output

• There are ways to interact with it, but it’s all done on the outside
  – Interaction possibilities are well defined
Encapsulation

• What is the right *concept* for each particular black box?

• Concepts are discovered through abstraction

• **Abstraction**: taking away inessential features, until only the essence of the concept remains
  – How much does a user really need to know?
Example: Cars

- Black boxes in a car: transmission, electronic control module, etc.

*Figure 1*
Levels of Abstraction in Automotive De
Example: Car

• 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 magically produced by an electronics component manufacturer
Example: Car

• 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
Example: Doorbell

• What can we do to a doorbell?

• What output do we get from a doorbell

• Do we actually know how a doorbell works?

• Do we need to?
Encapsulation

• In *Object-oriented programming* (OOP) the black boxes we use in our programs are objects
  - *String* is a black box
  - *Rectangle* is a black box
Software Design
Encapsulation

• Old times: computer programs only manipulated primitive types such as numbers and characters

• Gradually programs became more complex, vastly increasing the amount of detail a programmer had to remember and maintain
Encapsulation

• 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

• Encapsulation: Programmer using an object knows about its behavior, but not about its internal structure

• In software design, you can design good and bad abstractions / objects; understanding what makes good design is an important part of the education of a software engineer
Abstraction

• Who designs objects you use?
  – Other programmers

• What do these objects contain?
  – Other objects

• Lesson: Multiple levels of abstraction

• Almost always, you will be designing an abstraction while simultaneously using another one
Review

• Until now, you’ve only *used* objects
  – Analogous to engineer who puts final parts together in car

• Now you will learn how to *design* objects
  – Analogous to designing the parts of the car

• This is a complex process
Designing a Class

• Recall that a class is a template for objects
  – It’s the cookie cutter, not the cookies…

• You need to decide:
  – What does the black box need to do? (methods)
  – What does the black box need to know? (data)
Designing a Class

• Remember: you are designing a class that another programmer could use.
  – Make it easy to understand
  – Design your class as a black box
Methods

• The first thing a class requires is a list of methods – what should the black box do?

• For our Path example:
  - calcDistance
  - calcAngle
  - setX
  - setY
  - getX
  - getY

• Which methods are accessors and which are mutators?
Methods

- *access specifier* (eg *public*)
- *return type* (eg *String* or *void*)
- method name (eg *deposit*)
- list of *parameters* (eg New value for *x*)
- method *body* in braces {…}
Methods

• Examples:

    public double calcDistance() { ... }
    public int getX() { ... }
    public void setY(int newY) { ... }
Methods: Syntax

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

Example:
    public double calcDistance(){
        . . .
    }

Purpose:
• To define the behavior of a method
Methods

• Notes:
  – Methods *always* have parentheses
  – The *return type* is what type of output the black box will give the user
  – *Parameters* are what types of input the user needs to provide
  – The method body is a sequence of instructions
Constructors

• Recall that we need to be able to create objects in order to use them
• The set of instructions that does is this a constructor
• Note: Constructor name = Class name
  ```java
  public Path()
  {
      // body--filled in later
  }
  ```
Constructors

- Constructor body is executed when a new object is created
- Statements in constructor body will set up the object so it can be used
- A constructor initializes all data members
- How does the compiler know which constructor to call?
Constructors vs. Methods

• Constructors are a specialization of methods
  – Goal: to set the internal data of the object to a valid state
• 2 major differences
  – All constructors are named after the class
    • Therefore all constructors of a class have the same name
  – No return type is EVER listed!!
Constructors: Syntax

\texttt{accessSpecifier \textit{ClassName}(parameterType parameterName, \ldots)}

\{ 
  \textit{constructor body} 
\}

Example:

\texttt{public Path(int x, int y)}
\{ 
  \ldots 
\}

Purpose:

- To define the behavior of a constructor
Public Interface

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

```java
public class Path {
    // data fields--filled in later

    // Constructors
    public Path() {
        // body--filled in later
    }
}
```
public Path(int initX, int initY) {
    // body--filled in later
}

// Methods
public double calcDistance() {
    // body--filled in later
}

public int getX() {
    // body--filled in later
}

public void setY() {
    // body--filled in later
}

// more methods...
}
Class Definition Syntax

accessSpecifier class ClassName {
  fields
  constructors
  methods
}

Example:

public class Path {
  public Path(int initY, int initY) {...}
  public double calcDistance() {...}
  ...
}

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

• Public methods and constructors provide the public interface to a class
  – They are how you interact with the black box

• Our class is simple, but we can do many things with it
  – Notice we haven’t defined the method body yet, but know how we can use it
Javadoc Comments

• Part of creating a well-defined public interface is always **commenting** the class and method behaviors

• The HTML pages from the API are created from special comments in your program called *javadoc comments*

• Placed *before* the class or method
  
  ```java
  /** ... */
  ```
Javadoc Comments

• Begin with /**
• Ends with */
• Put * on lines in between as convention, makes it easier to read
• **Javadoc tags** - @ indicates a tag
  – @author, @param, @return
• Benefits
  – Online documentation
  – Other java documents
Javadoc Comments

• First part is a description of the method
  – Carefully explain method

• @param for each parameter
  – Omit if no parameters

• @return for the value returned
  – Omit if return type void
/**
 * Calculates line of sight distance
 * @return the calculated distance
 */

public double calcDistance(){
    // implementation filled in later
}

/**
 * Changes the vertical distance
 * @param newY the new distance in the y direction
 */

public void setY(int newY){
    // implementation filled in later
}
Javadoc Comments

• A class comment succinctly describes the class

```java
/**
 * A path has a vertical distance and horizontal distance and users can determine diagonal distance and angles using this class
 * @author David Koop (dakoop)
 */

class Path {
    ...
}
```
Commenting

• Seems repetitive, but is necessary
  – Very helpful if you comment before you code your methods

• Provide documentation comments for
  – every class
  – every method
  – every parameter
  – every return value.

• When in doubt, comment
  – But make sure comments are concise
Figure 3  A Method Summary Generated by javadoc
**Figure 4**  Method Detail Generated by javadoc

```java
public void deposit(double amount)

Deposits money into the bank account.

**Parameters:**

amount - the amount to deposit
```
Instance Fields

• Remember: methods and constructors comprise the **public** interface of a class

• Instance Fields are part of the internal workings - 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
  – AKA Data Members
Instance Fields

• The class declaration specifies the instance fields:

```java
public class Path {
    private int xDistance;
    private int yDistance;
    ...
}
```
Instance Fields

• An instance field declaration consists of the following parts:
  – access specifier (usually private)
  – type of variable (eg double)
  – name of variable (eg xDirection)

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

• You should declare all instance fields as private
Access Specifiers

- **Access Specifier** – defines the accessibility of the instance field and methods
  - *private* – only accessible within methods defined in the class
  - *public* – accessible both within methods defined in the class and in other classes

- *private* enforces encapsulation/black box

- **AKA Visibility Modifier**
harrysChecking = BankAccount
  balance = 

momsSavings = BankAccount
  balance = 

Figure 5  Instance Fields
Instance Fields

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

**Example:**
```java
public class Path
{
   . . .
   private int xDistance;
   . . .
}
```

**Purpose:** To define a field that is present in every object of a class
Accessing Instance Fields

• The `setX` method of the `Path` class can access the private instance field `xDistance`:

```java
public void setX(int newX) {
    xDistance = newX;
}
```
Accessing Instance Fields

• Other methods cannot access `xDistance`:

```java
public class ChangePath {
    public static void main(String[] args) {
        Path p = new Path(3,4);
        ...
        p.xDistance = 12; // ERROR
    }
}
```
Instance Fields

- Encapsulation = *Hiding* data and providing access through methods

- By making data members private, we hide internal workings of a class from a user

- Note: We can have public instance fields and private methods, but commonly we do not
Constructor Bodies

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

```java
public Path()
{
    xDistance = 0;
    yDistance = 0;
}

public Path(int initX, int initY)
{
    xDistance = initX;
    yDistance = initY;
}
```
Constructors – What Happens?

Path p = new Path(4,5);

– Create a new object of type Path
– Call the second version of the constructor (since parameters are supplied)
  • Sets the parameters initX to 4 and initY to 5
  • Executes the list of instructions in the constructor body
– 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 variable p
Method Bodies

- Methods have several components that allow them to be functional units of code.
  - Declaration (method signature)
  - Method Body
    - Makes use of parameters (inputs)
    - May return a value (output)
    - Sequence of instructions – may call other methods, declare variables, create objects, print output, etc.
Method Bodies

• Some methods do not return a value
  public void setY(int newY)
  {
    yDistance = newY;
  }

• Some methods return an output value
  public int getX()
  {
    return xDistance;
  }
public double calcDistance()
{
    double sumSquares = xDistance*xDistance +
                        yDistance*yDistance;
    return Math.sqrt(sumSquares);
}
Methods – What Happens?

• `p.setY(87);`
  – Set the parameter variable `newY` to 500
  – Executes the list of instructions in the method body:
    • Sets the instance field `yDistance` to value of `newY`
• `p.getX();`
  – No parameter values to set
  – Executes the list of instructions in the method body
  – Returns the value of expression after `return` statement
Return Statement Syntax

\[
\text{return expression;}
\]
\[
\text{or}
\]
\[
\text{return;}
\]

Example:

\[
\text{return balance;}
\]

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.
/**
 * A path has a vertical distance and horizontal distance and users can
 * determine diagonal distance and angles using this class
 * @author David Koop (dakoop)
 */

public class Path {

    private int xDistance;
    private int yDistance;

    /**
     * Constructs a path with distance zero in both the x and y direction
     *
     */
    public Path() {
        xDistance = 0;
        yDistance = 0;
    }

    /**
     * Constructs a path with distances initX and initY in the x and y
     * directions, respectively
     * @param initX initial x distance
     * @param initY initial y distance
     */
    public Path(int initX, int initY) {
        xDistance = initX;
        yDistance = initY;
    }
}
/**
 * Returns the distance in the x direction
 * @return distance in the x direction
 */
public int getX() {
    return xDistance;
}

/**
 * Returns the distance in the y direction
 * @return distance in the y direction
 */
public int getY() {
    return yDistance;
}

/**
 * Changes the distance in the x direction
 * @param newX distance in the x direction
 */
public void setX(int newX) {
    xDistance = newX;
}

/**
 * Changes the distance in the y direction
 * @param newY distance in the y direction
 */
public void setY(int newY) {
    yDistance = newY;
}
/**
 * Calculates the line of sight distance
 * @return the calculated distance
 */

public double calcDistance() {
    double sumSquares = xDistance*xDistance + yDistance*yDistance;
    return Math.sqrt(sumSquares);
}

/**
 * Calculates the angle of the line of sight
 * @return the calculated angle, in radians
 */

public double calcAngle() {
    return Math.atan(((double) yDistance) / (double) xDistance);
}
Methods

• Why do we write methods? Why not just have one long list of instructions in our programs?

1. smaller $\leftrightarrow$ easier
2. test & debug once, execute as often as needed
3. much more readable code
Three Types of Classes

• This isn’t in the book…
• In this course, there are three types of classes you will write:
  1. A new encapsulation
  2. A tester class
  3. An application
Classes: Type 1

• Source code that defines a new data type by encapsulating:
  – data members
  – constructors
  – methods

• Ex. Path class we just created

• AKA **Instantiable class** – designed so that we can make objects
/** Stores information about one pet. */
public class Pet {

Data member
/** Name of the pet. */
private String name, kind;
private int age;

Constructor
/** Initializes a new instance.
 * @param n The pet's name.
 */
public Pet( String n, String k, int a ) {
    name = n; kind = k; age = a;
}

Methods
/** Returns the name of this pet. */
public String getName() { return name; }

/** Changes the name of this pet.
 * @param newName The new name of the pet.
 */
public void changeNameTo( String newName ) {
    // TEST THE NAME HERE
    name = newName;
}
Classes: Type 2

- Code that is written to test another class.

- Has a main method that:
  1. Creates instances of the class
  2. Executes methods
  3. Compares the actual output with expected output

- Classes should be tested before being used in Java Applications.
/** A Test Program. */
public class TestPet {

/**
 * Test Program starts here.
 * @param args Unused in this program.
 */
public static void main( String[] args ) {

// Declare and create a pet
Pet pet1 = new Pet( "George", "bird", 14 );

// Test the pet's information
testGetName( pet1, "George" );

/** Tests Pet getName method. */
public static void testGetName( Pet pet, String expectedName ) {
    System.out.println( pet.getName() + " should be " + expectedName );
}
}
Classes: Type 3

- Java Application

- Source code that uses instances of other classes (objects) to solve problems.

- *Must* have a *main* method.

- Ex. Programs you completed in A0
public class PetApplication {

/**
 * Program starts here.
 * @param args Unused in this program.
 */

public static void main( String[] args ) {

    // Declare and create a pet
    Pet pet1 = new Pet( "George", "bird", 14 );

    // Output the pet's information
    display( pet1 );
}

/**
 * Prints Pet information to screen.
 */

public static void display( Pet pet ) {
    System.out.println(
        pet.getName() + " is a " +
        pet.getAge() + " year old " +
        pet.getKind() );
}
}
Testing Instantiable Classes

• We’ve seen how to design an instantiable class, but…

• By itself, we cannot actually run an instantiable class
  – It has no `main()` method, like most classes

• We create instances of it in other classes

• Idea: use a test class to make sure it works properly before letting people use our class
Testing Instantiable Classes

• 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
  – Verify output is correct and the program behaves as expected
Testing Instantiable Classes

• Details for building the program vary. In most environments, you need to carry out these steps:
  – Write a test program for the methods
  – Include the class being tested in the project
  – Run the test program
/**
 * A class to test the Path class.
 */

public class PathTester{
    /**
     * Tests the methods of the Path class.
     * @param args not used
     */
    public static void main(String[] args){
        Path p = new Path(3,6);
        p.setY(4);
        System.out.println(p.calcDistance());
        p.setX(2);
        System.out.println(p.calcDistance());
    }
}
Categories of Variables

• Categories of variables
  – Instance fields (\texttt{xDistance} in \texttt{Path} class)
  – Local variables (\texttt{sumSquares} in \texttt{calcDistance} method)
  – Parameter variables (\texttt{newY} in \texttt{setY} method)

• Share similar routines in terms of declaring and creating
• Differ in their \textbf{lifetime (AKA scope)}
Instance Fields

- An instance field belongs to an object
  - AKA Instance variable, data member
  - Each object has its own copy of the instance field
- The fields stay alive until no method uses the object any longer
- More specifically, until the object no longer exists
- Note: Instance fields have access specifiers, others don’t
Garbage Collector

• In Java, the garbage collector periodically reclaims objects when they are no longer used
  – If no variables reference the object anymore, it can no longer be used and is collected
Local Variables

• Local and parameter variables belong to a method
  – You declare them and create within the method

• When method is done executing, variable is thrown out
  – **EVERY TIME** the method is executed, a new copy of any variables is created, values are **NOT SAVED**
Parameter variables

- Parameter variables are initialized to the values that are passed to them

- Maintain same lifetime as local variables

- Only difference is that they receive values “automatically” when a method is called
Scope

• Scope is where a variable “lives” in some sense – what it’s lifetime is

• For example, a local variable or parameter can only live in the method it’s defined in

• An instance field can be accessed by any of an object’s methods and stays around as long as the object does
Scope

• Compare the lifetimes of \texttt{sumSquares}, \texttt{xDistance}, and \texttt{newY}

• It’s not important to store everything forever!

• Instance fields only need store that which you can’t do without
Scope

- Instance fields are initialized to a default value, but you must initialize local variables
  - Default values for numbers is 0
  - Objects is `null`
  - COMMON ERROR: forgetting to create objects for instance fields, declaration leaves it at `null`

- Not an issue with parameter variables – Why?
The Implicit Parameter

• The *implicit parameter* of a method is the object on which the method is invoked

• Why is this important?

• When a method is invoked, and an instance field is used, how does the computer know which object it belongs to?
The Implicit Parameter

• Use of an instance field name in a method denotes the instance field of the object

```java
public double calcDistance()
{
    double sumSquares=xDistance*xDistance + yDistance*yDistance;
    return Math.sqrt(sumSquares);
}
```
The Implicit Parameter

• xDistance is the xDistance of the object to the left of the dot:

p.calcDistance()

means

double sumSquares = p.xDistance*p.xDistance + p.yDistance*p.yDistance;
return Math.sqrt(sumSquares);
The Implicit Parameter

• *Every* method has one implicit parameter

• The implicit parameter is always called `this`
  – The method knows what object called it based on a reference, but does not know/care about the identifier name
  – E.g. you cannot say `p.xDistance` in the class, because `p` is a local variable of another class

• Exception: Static methods do not have an implicit parameter (more in Chapter 9)
Implicit Parameters and \textit{this}

double \textit{sumSquares} = \textit{xDistance}^2 + \textit{yDistance}^2;

// actually means

double \textit{sumSquares} =
\textit{this.xDistance}^2 + \textit{this.yDistance}^2;

\begin{itemize}
  \item When you refer to an instance field in a method, the compiler automatically applies it to the \textit{this} parameter
  \end{itemize}

\texttt{p.calcDistance();}
The Implicit Parameter

Figure 8   The Implicit Parameter of a Method Call
Advanced Info

• **this is very powerful**

• Can call one constructor from another to avoid redundancy
  – Important goal in programming, never have duplicate code
public class Path{
    private int xDistance;
    private int yDistance;

    public Path(int initX, int initY){
        xDistance = initX;
        yDistance = initY;
    }

    public Path() {
        xDistance = 0;
        yDistance = 0;
    }

    ...
}

public class Path{
    private int xDistance;
    private int yDistance;
    
    public Path(int initX, int initY){
        xDistance = initX;
        yDistance = initY;
    }
    
    public Path() {
        this(0,0);
    }
    ...
}

Recap on Designing Classes

• Step 1: Find out what you are asked to do with an object of the class
  – What do you want to be able to do with the object?

• Step 2: Specify the Public Interface
  – Convert the list from Step 1 into methods, and determine the parameters (think input) they need
  – Constructors: how do you want to be able to create objects?
Recap on Designing Classes

• Step 3: Document the public interface
  – Create Javadoc comments for each method and the class

• Step 4: Determine instance fields
  – What information needs to be maintained?
  – Don’t keep extra information

• Step 5: Implement
  – One at a time, from easiest to most difficult
Recap on Designing Classes

• Go back to Step 2 if implementation doesn’t work

• Step 6: Test your class
Notes on Style

• A couple ways to use curly braces

Option #1
public class SomeClass {
  ...
}

Option #2
public class SomeClass
{
  ...
}
Notes on Style

• Book says to place instance fields at the bottom of a class

```java
public class SomeClass{
    //constructors
    //methods
    //instance fields
}
```

• Most conventions have it at the top of a class

```java
public class SomeClass{
    //instance fields
    //constructors
    //methods
}
```
Notes on Style

• There are variations on import – either of these three will work
  – Don’t use import (fully qualify all classes)
  – Import just one class (java.awt.Rectangle)
  – Import the entire package (java.awt.*)
Notes on Style

• Is there a correct way to style your code?
• The computer doesn’t care…

• You MUST be consistent
  – Don’t use both styles of curly braces in the same program
  – Don’t put instance fields at the top and bottom of your class declaration