Programming in Java

with BlueJ

BlueJ

The Modern “Smalltalk”
IDE for JAVA

BlueJ

www.bluej.org

Bluej

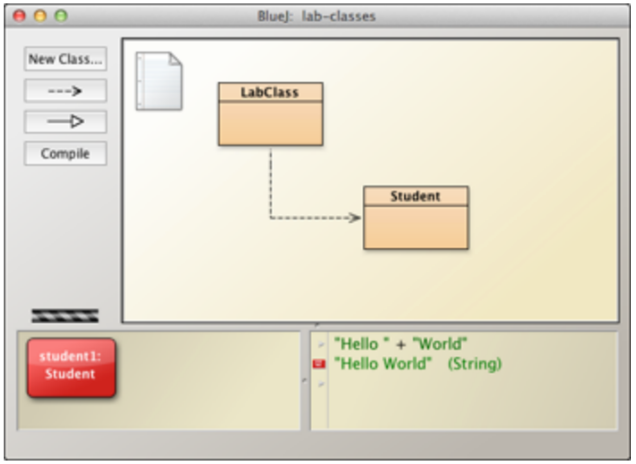
LIBRARY WEB ADMIN CODE EDU LABS SJSU 202 281 279 DOCS ZAPP BOOKS

BlueJ

A free Java Development Environment designed for beginners, used by millions worldwide. Find out more...

"One of my favourite IDEs out there is BlueJ"

— James Gosling, creator of Java.

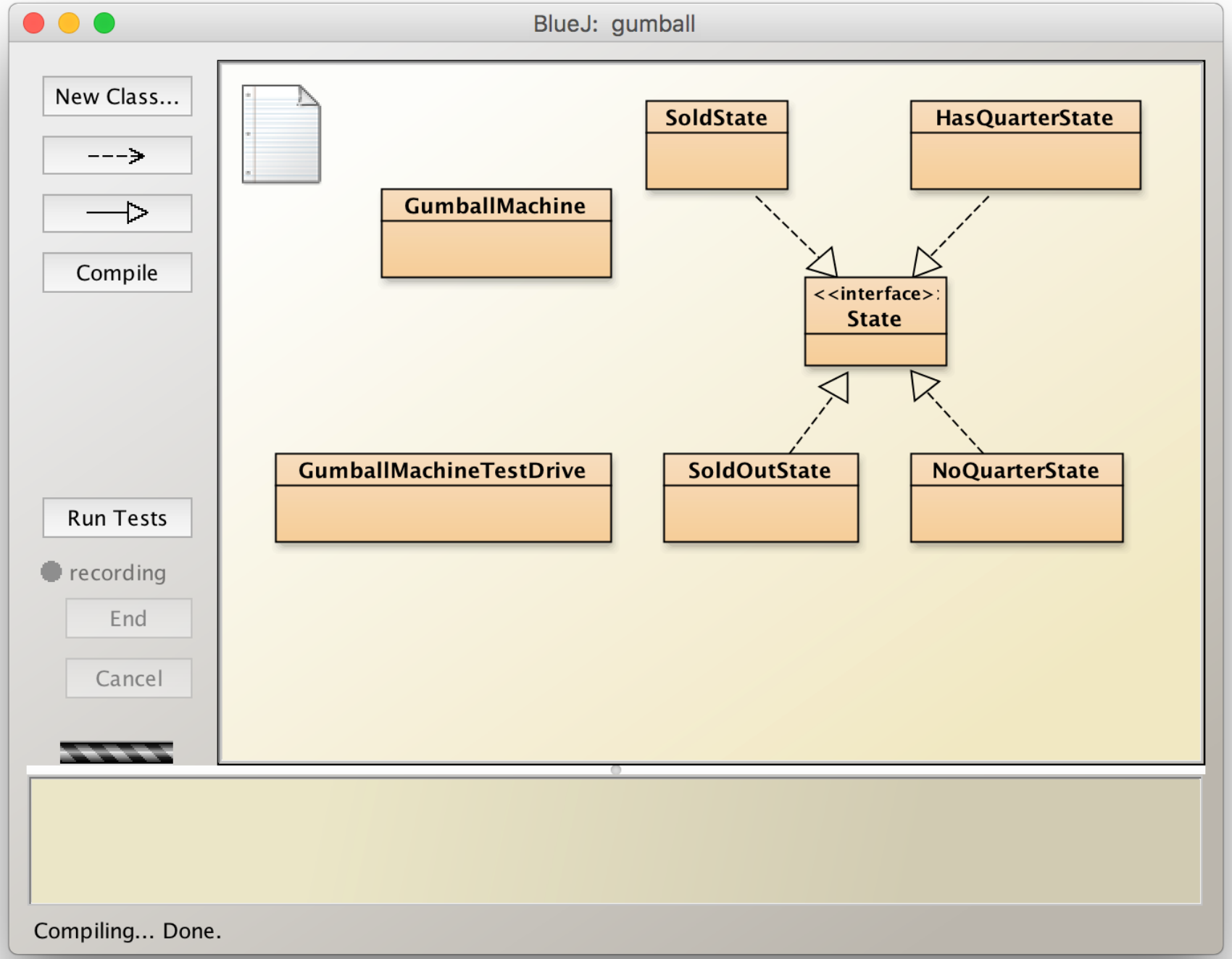


Created by University of Kent

Supported by ORACLE

Download and Install

Version 3.1.7, released 23 February 2016 (what's new)


BlueJ: gumball
New Class...
Compile
Run Tests
recording
End
Cancel
SoldState
HasQuarterState
GumballMachine
<<interface>>
State
GumballMachineTestDrive
SoldOutState
NoQuarterState
Compiling... Done.

Primitive Types & Autoboxing

Primitive type	Wrapper type
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Primitive types

The following table lists all the primitive types of the Java language:

Type name	Description	Example literals		
Integer numbers				
byte	byte-sized integer (8 bit)	24	–2	
short	short integer (16 bit)	137	–119	
int	integer (32 bit)	5409	–2003	
long	long integer (64 bit)	423266353L	55L	
Real numbers				
float	single-precision floating point	43.889F		
double	double-precision floating point	45.63	2.4e5	
Other types				
char	a single character (16 bit)	'm'	'?'	'\u00F6'
boolean	a boolean value (true or false)	true	false	

Whenever a value of a primitive type is used in a context that requires an object type, the compiler uses autoboxing to automatically wrap the primitive-type value in an appropriate wrapper object. This means that primitive-type values can be added directly to a collection, for instance. The reverse operation – *unboxing* – is also performed automatically when a wrapper-type object is used in a context that requires a value of the corresponding primitive type.

Control Structures

```
if(expression) {
    statements
}
else {
    statements
}
```

```
switch (expression) {
    case value1:
    case value2:
    case value3:
       statements;
       break;
    case value4:
    case value5:
       statements;
       break;
    further cases omitted
    default:
       statements;
       break;
}
```

```
while(expression) {
      statements
}
```

```
do {
    statements
} while(expression);
```

```
for(variable-declaration: collection) {
    statements
}
```

```
for(String note: list) {
    System.out.println(note);
}
```

```
for(int i = 0; i < text.size(); i++) {
    System.out.println(text.get(i));
}
```

```
try {
    statements
}
catch(exception-type name) {
     statements
}

finally {
    statements
}
```

Class Definitions

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * Instances will check to ensure that a user only enters
 * sensible amounts of money, and will only print a ticket
 * if enough money has been input.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2008.03.30
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;

    /**
     * Create a machine that issues tickets of the given price.
     */
    public TicketMachine(int ticketCost)
    {
        price = ticketCost;
        balance = 0;
        total = 0;
    }

    /**
     * @Return The price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }

    /**
     * Return The amount of money already inserted for the
     * next ticket.
     */
    public int getBalance()
    {
        return balance;
    }
```

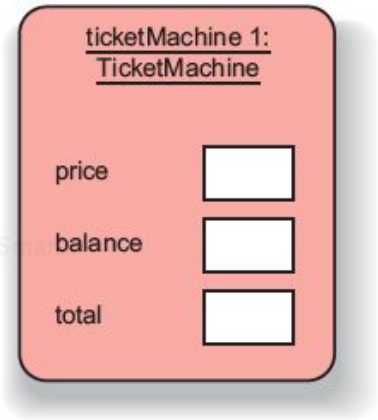
```
    /**
     * Receive an amount of money in cents from a customer.
     * Check that the amount is sensible.
     */
    public void insertMoney(int amount)
    {
        if(amount > 0) {
            balance = balance + amount;
        }
        else {
            System.out.println("Use a positive amount: " +
                               amount);
        }
    }

    /**
     * Print a ticket if enough money has been inserted, and
     * reduce the current balance by the ticket price. Print
     * an error message if more money is required.
     */
    public void printTicket()
    {
        if(balance >= price) {
            // Simulate the printing of a ticket.
            System.out.println("##################");
            System.out.println("# The BlueJ Line");
            System.out.println("# Ticket");
            System.out.println("# " + price + " cents.");
            System.out.println("##################");
            System.out.println();

            // Update the total collected with the price.
            total = total + price;
            // Reduce the balance by the prince.
            balance = balance - price;
        }
        else {
            System.out.println("You must insert at least: " +
                               (price - balance) + " more cents.");

        }
    }
}
```

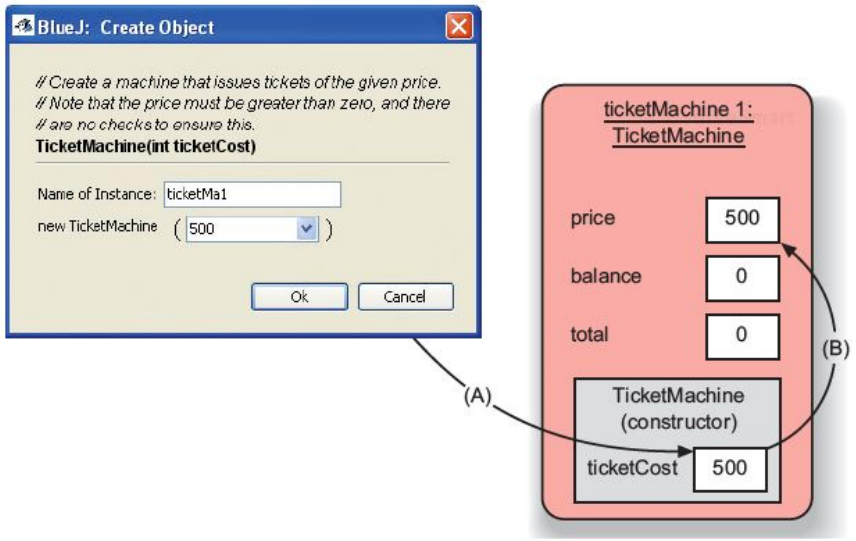
Fields & Constructors

```
public class ClassName
{
    Fields
    Constructors
    Methods
}
```

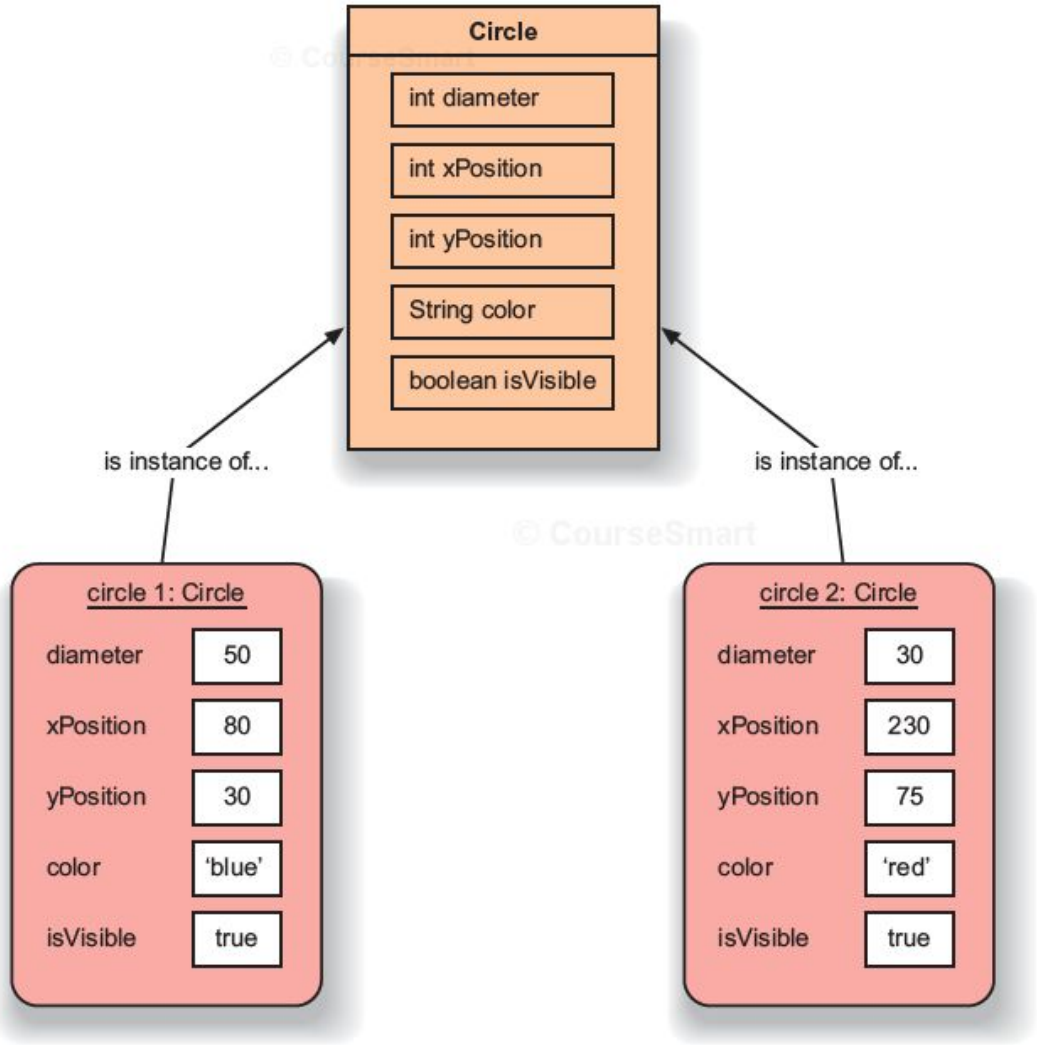
```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;

    Constructor and methods omitted.
}
```

```
public class TicketMachine
{
    Fields omitted.

    /**
     * Create a machine that issues tickets of the given price.
     * Note that the price must be greater than zero, and there
     * are no checks to ensure this.
     */
    public TicketMachine(int ticketCost)
    {
        price = ticketCost;
        balance = 0;
        total = 0;
    }

    Methods omitted.
}
```





Objects as Instances



Accessors

Concept:

Accessor methods return information about the state of an object.

```
public class TicketMachine
{
    Fields omitted.

    Constructor omitted.

    /**
     * Return the price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }

    Remaining methods omitted.
}
```

Mutators

```
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;
```

Concept:

Mutator methods change the state of an object.

```
/**
 * Receive an amount of money in cents from a customer.
 */
public void insertMoney(int amount)
{
    balance = balance + amount;
}
```

Printing to the Screen

Concept:

The method **System.out.println** prints its parameter to the text terminal.

```
/**
 * Print a ticket and reduce the
 * current balance to zero.
 */
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Update the total collected with the balance.
    total = total + balance;
    // Clear the balance.
    balance = 0;
}
```

Abstraction

Concept:

Classes define types. A class name can be used as the type for a variable. Variables that have a class as their type can store objects of that class.

The clock example


```
public class NumberDisplay
{
    private int limit;
    private int value;

    Constructor and methods omitted.
}
```

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Constructor and methods omitted.
}
```

Concept:

Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.

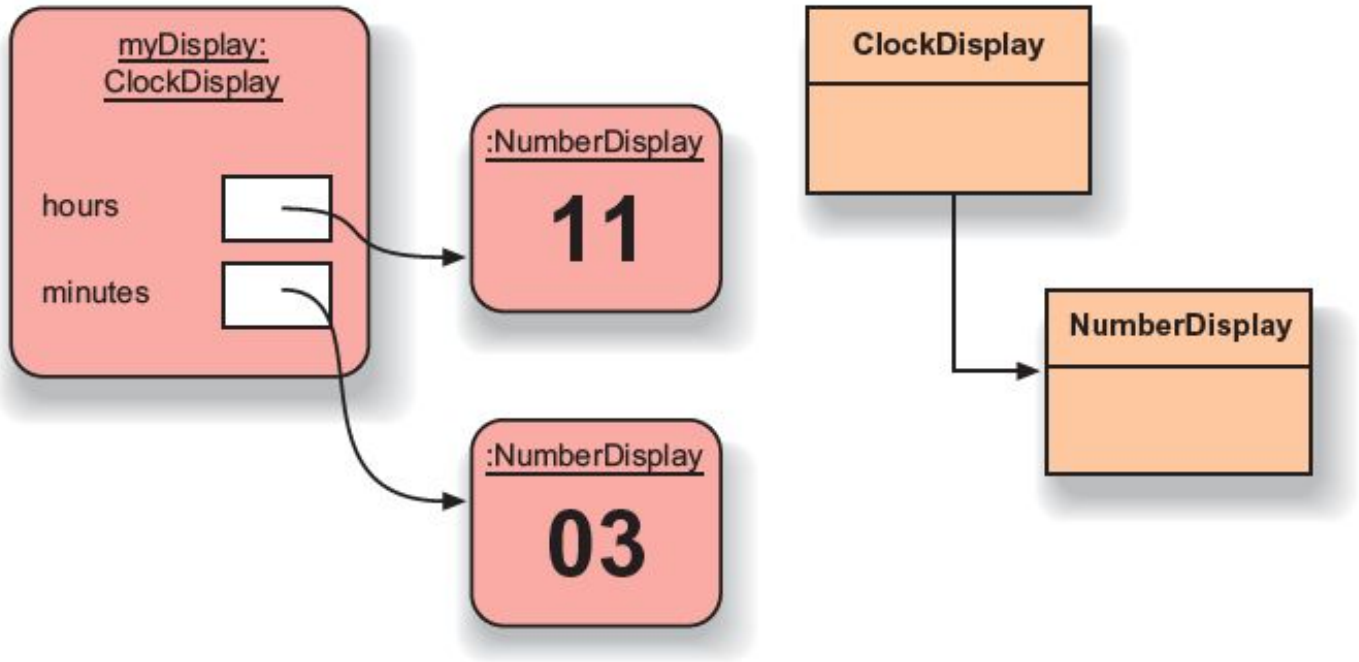
Concept:

Modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

Classes versus Objects

Concept:

The **class diagram** shows the classes of an application and the relationships between them. It gives information about the source code. It presents the static view of a program.



The Clock Display

```
public class NumberDisplay
{
    private int limit;
    private int value;
```

```
    /**
     * Constructor for objects of class NumberDisplay
     */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

    /**
     * Return the current value.
     */
    public int getValue()
    {
        return value;
    }

    /**
     * Set the value of the display to the new specified
     * value. If the new value is less than zero or over the
     * limit, do nothing.
     */
    public void setValue(int replacementValue)
    {
        if((replacementValue >= 0) &&
                (replacementValue < limit)) {
            value = replacementValue;
        }
    }
```

The clock example

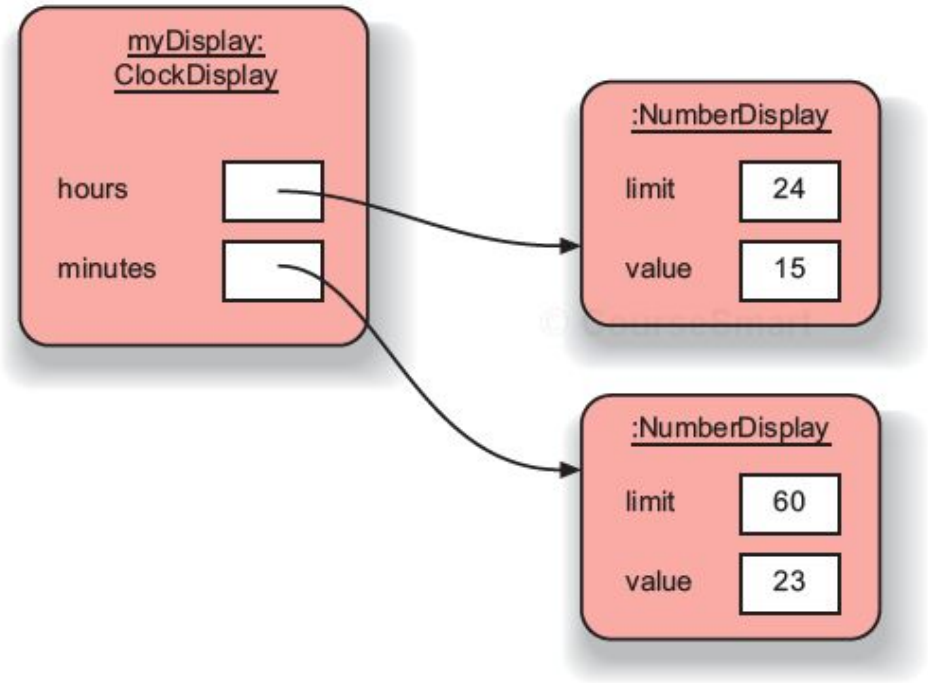

```
    /**
     * Increment the display value by one, rolling over to zero if
     * the limit is reached.
     */
    public void increment()
    {
        value = (value + 1) % limit;
    }
```

```
    /**
     * Return the display value (that is, the current value
     * as a two-digit String. If the value is less than ten,
     * it will be padded with a leading zero).
     */
    public String getDisplayValue()
    {
        if(value < 10) {
            return "0" + value;
        }
        else {
            return "" + value;
        }
    }
```

Objects Creating Objects

Concept:

Object creation. Objects can **create** other **objects** using the new operator.




```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Remaining fields omitted.

    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    Methods omitted.
}
```

Overloading

Concept:

Overloading. A class may contain more than one constructor, or more than one method of the same name, as long as each has a distinctive set of parameter types.

```
new ClockDisplay()
new ClockDisplay(hour, minute)
```

Method Calls

Concept:

Methods can call methods of other objects using dot notation. This is called an **external method call**.

object.method()

```
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) { // it just rolled over!
        hours.increment();
    }
    updateDisplay();
}
```

Concept:

Methods can call other methods of the same class as part of their implementation. This is called an **internal method call**.

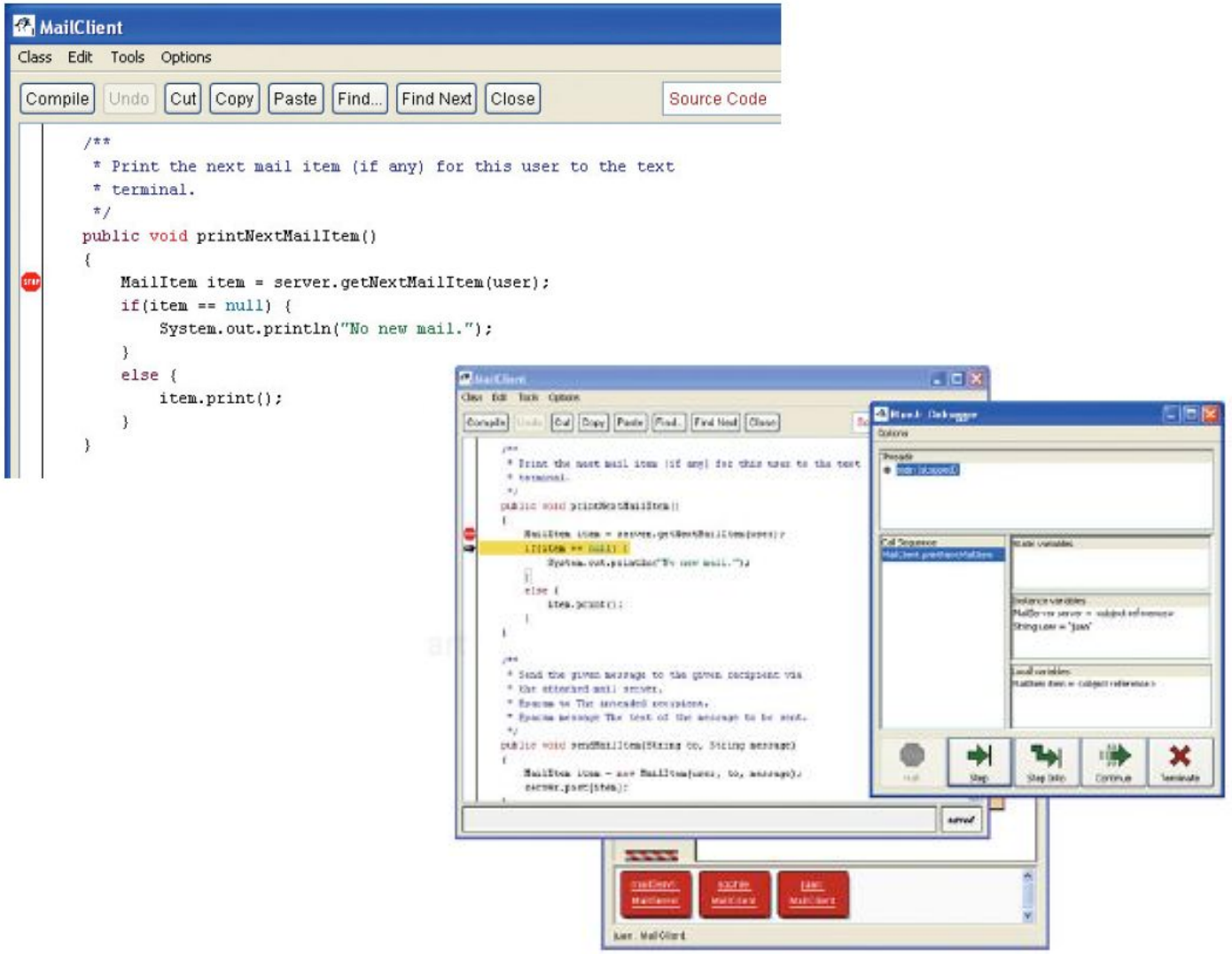
Implied “this” reference. i.e. this.updateDisplay()

“this” keyword

```
public class MailItem
{
    // The sender of the item.
    private String from;
    // The intended recipient.
    private String to;
    // The text of the message.
    private String message;

    /**
     * Create a mail item from sender to the given recipient,
     * containing the given message.
     * @param from    The sender of this item.
     * @param to      The intended recipient of this item.
     * @param message The text of the message to be sent.
     */
    public MailItem(String from, String to, String message)
    {
        this.from = from;
        this.to = to;
        this.message = message;
    }

    Methods omitted.
}
```

Debugging in BlueJ

Set Breakpoint



Using Java SDK

http://download.oracle.com/javase/7/docs/api/

```
import java.util.ArrayList;

/**
 * A class to maintain an arbitrarily long list of notes.
 * Notes are numbered for external reference by a human user.
 * In this version, note numbers start at 0.
 *
 * @author David J. Barnes and Michael Kölling.
 * @version 2008.03.30
 */
public class Notebook
{
    // Storage for an arbitrary number of notes.
    private ArrayList<String> notes;

    /**
     * Perform any initialization that is required for the
     * notebook.
     */
    public Notebook()
    {
        notes = new ArrayList<String>();
    }

    /**
     * Store a new note into the notebook.
     * @param note The note to be stored.
     */
    public void storeNote(String note)
    {
        notes.add(note);
    }

    /**
     * @return The number of notes currently in the notebook.
     */
    public int numberOfNotes()
    {
        return notes.size();
    }
```

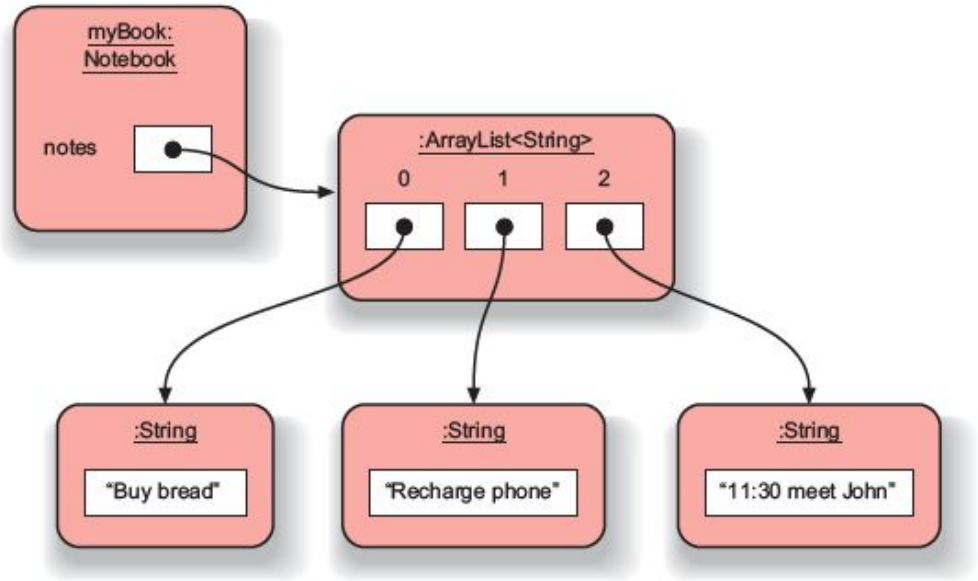
Import Class Definition

Now, can say:
ArrayList
instead of
java.util.ArrayList

```
    /**
     * Show a note.
     * @param noteNumber The number of the note to be shown.
     */
    public void showNote(int noteNumber)
    {
        if(noteNumber < 0) {
            // This is not a valid note number, so do nothing.
        }
        else if(noteNumber < numberOfNotes()) {
            // This is a valid note number, so we can print it.
            System.out.println(notes.get(noteNumber));
        }
        else {
            // This is not a valid note number, so do nothing.
        }
    }
}
```

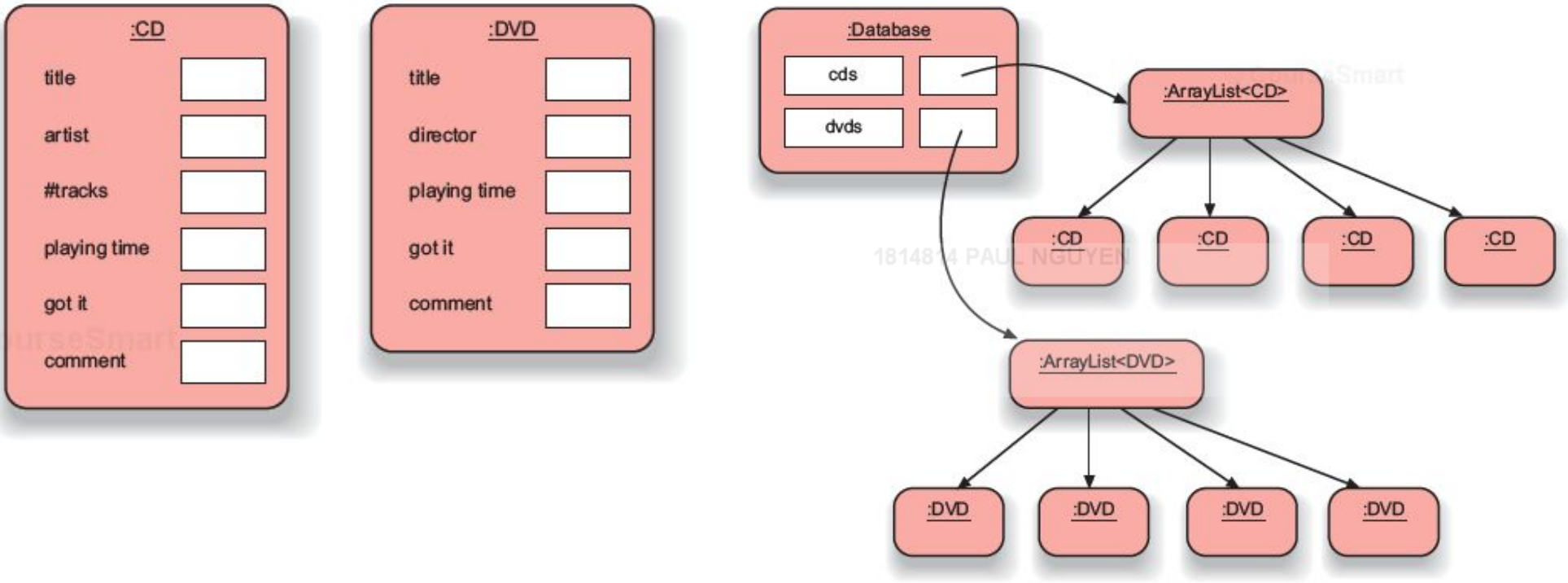
Adding & Removing from a List

```
    /**
     * Store a new note into the notebook.
     * @param note The note to be stored.
     */
    public void storeNote(String note)
    {
        notes.add(note);
    }
```

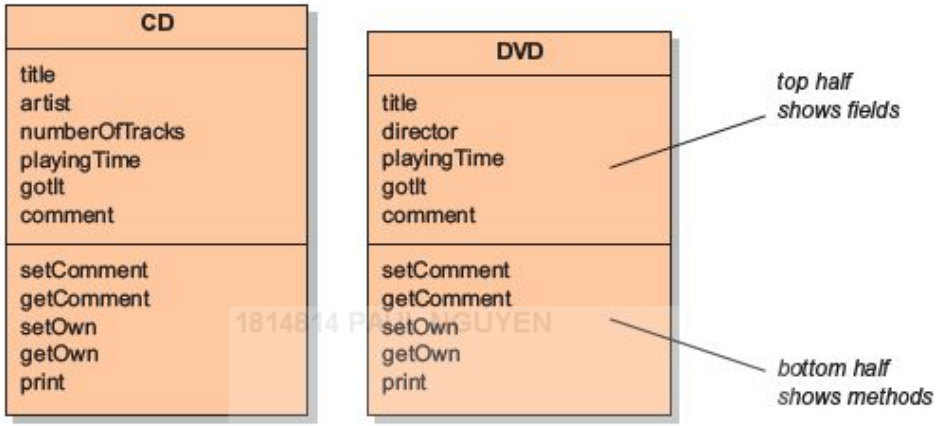



```
public void removeNote(int noteNumber)
{
    if(noteNumber < 0) {
        // This is not a valid note number, so do nothing.
    }
    else if(noteNumber < numberOfNotes()) {
        // This is a valid note number, so we can remove it.
        notes.remove(noteNumber);
    }
    else {
        // This is not a valid note number, so do nothing.
    }
}
```

Object Structure

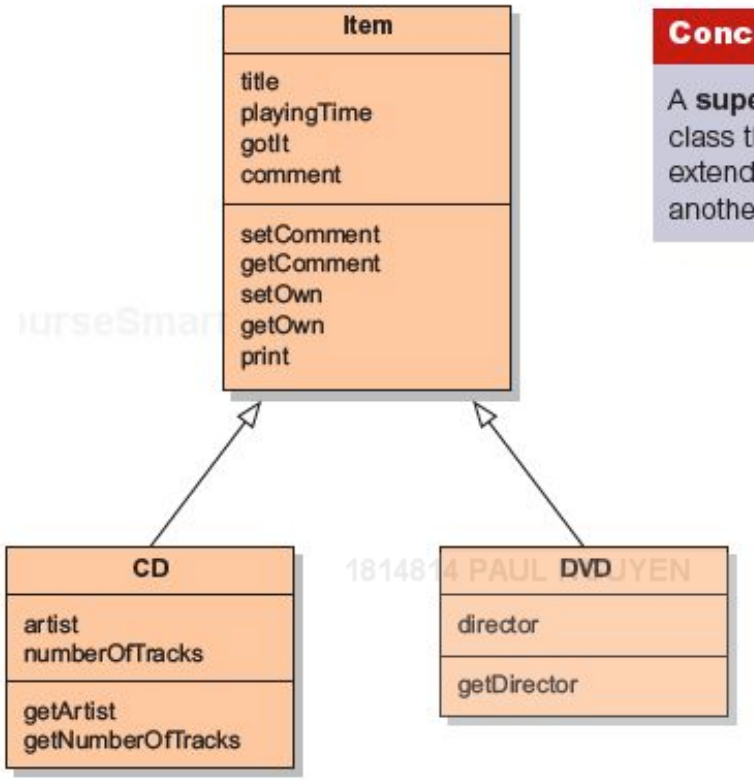


Using Inheritance



Concept:

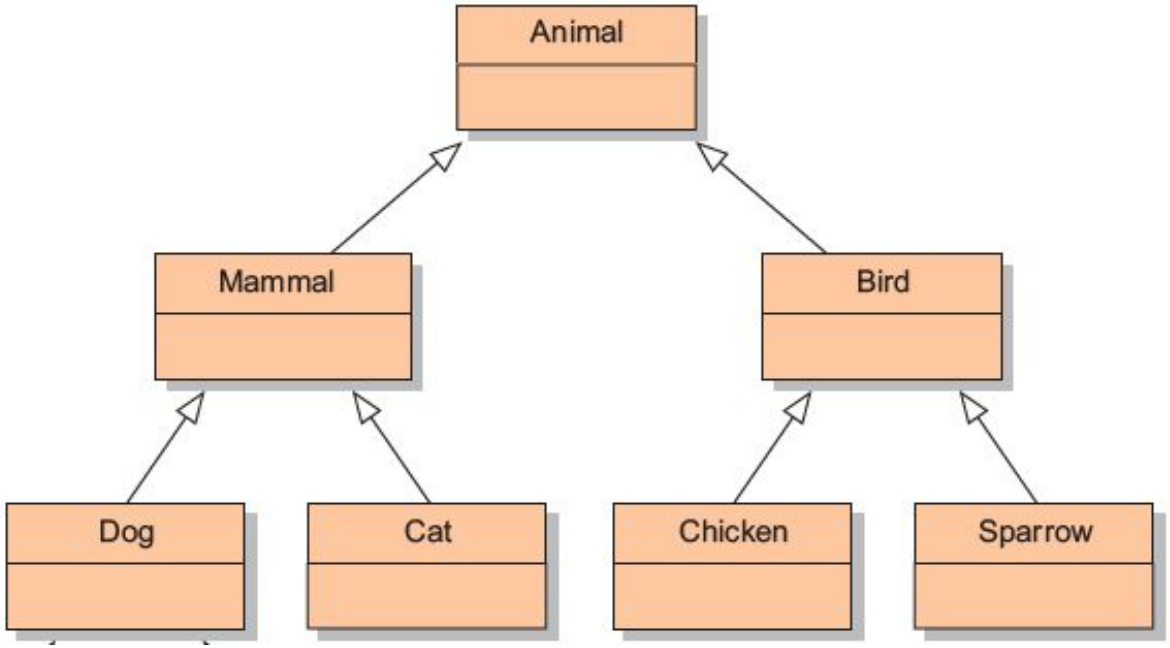
A **subclass** is a class that extends (inherits from) another class. It inherits all fields and methods from its superclass.



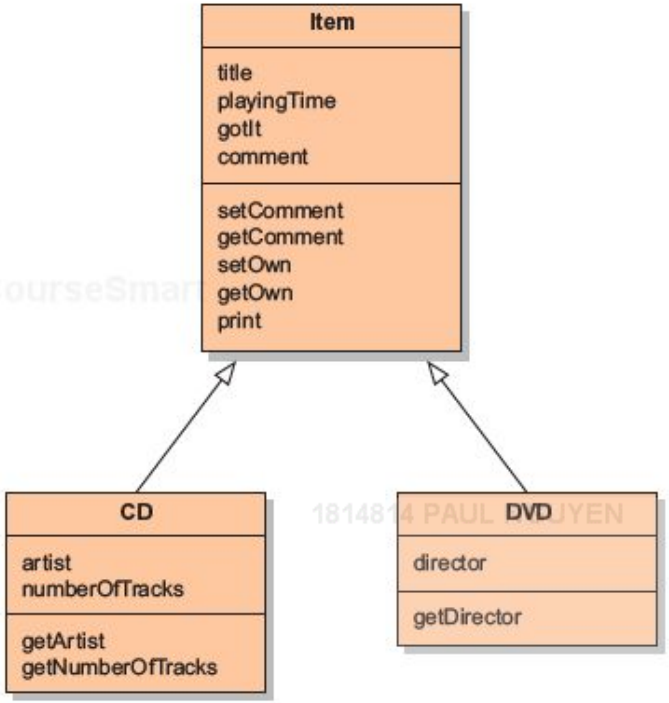
Concept:

A **superclass** is a class that is extended by another class.

Object Hierarchies in the real world



Inheritance in Java




```
public class Item
{
    private String title;
    private int playingTime;
    private boolean gotIt;
    private String comment;

    // constructors and methods omitted.
}
```

```
public class CD extends Item
{
    private String artist;
    private int numberOfTracks;

    // constructors and methods omitted.
}
```

```
public class DVD extends Item
{
    private String director;

    // constructors and methods omitted.
}
```

Abstract Classes

Concept:

An **abstract method** definition consists of a method signature without a method body. It is marked with the key word `abstract`.

Concept:

An **abstract class** is a class that is not intended for creating instances. Its purpose is to serve as a superclass for other classes. Abstract classes may contain abstract methods.

```
public abstract class Animal
{
    // fields omitted

    /**
     * Make this animal act – that is: make it do whatever
     * it wants/needs to do.
     * @param newAnimals A list to add newly born animals to.
     */
    abstract public void act(List<Animal> newAnimals);

    // other methods omitted
}
```

Concept:

Abstract subclass. For a subclass of an abstract class to become concrete, it must provide implementations for all inherited abstract methods. Otherwise the subclass will itself be abstract.

Interfaces

Concept:

A Java **interface** is a specification of a type (in the form of a type name and a set of methods) that does not define any implementation for the methods.

Note: Java does not support multiple inheritance

```
public class Fox extends Animal implements Drawable
{
    // Body of class omitted.
}
```

```
/**
 * The interface to be extended by any class wishing
 * to participate in the simulation.
 */
public interface Actor
{
    /**
     * Perform the actor's regular behavior.
     * @param newActors A list for storing newly created actors.
     */
    void act(List<Actor> newActors);

    /**
     * Is the actor still active?
     * @return true if still active, false if not.
     */
    boolean isActive();
}
```

```
public class Hunter implements Actor, Drawable
{
    // Body of class omitted.
}
```

Important Java Libraries http://download.oracle.com/javase/7/docs/api/

package `java.lang` – Summary of the most important classes	
interface `Comparable`	Implementation of this interface allows comparison and ordering of objects from the implementing class. Static utility methods such as `Arrays.sort` and `Collections.sort` can then provide efficient sorting in such cases, for instance.
class `Math`	`Math` is a class containing only static fields and methods. Values for the mathematical constants `e` and π are defined here, along with trigonometric functions, and others such as `abs`, `min`, `max`, and `sqrt`.
class `Object`	All classes have `Object` as a superclass at the root of their class hierarchy. From it all objects inherit default implementations for important methods such as `equals` and `toString`. Other significant methods defined by this class are `clone` and `hashCode`.
class `String`	Strings are an important feature of many applications, and they receive special treatment in Java. Key methods of the `String` class are `charAt`, `equals`, `indexOf`, `length`, `split`, and `substring`. Strings are immutable objects, so methods such as `trim` that appear to be mutators actually return a new `String` object representing the result of the operation.
class `StringBuilder`	The `StringBuilder` class offers an efficient alternative to `String` when it is required to build up a string from a number of components: e.g., via concatenation. Its key methods are `append`, `insert`, and `toString`.

package `java.util` – Summary of the most important classes and interfaces

interface `Collection`	This interface provides the core set of methods for most of the collection-based classes defined in the `java.util` package, such as `ArrayList`, `HashSet`, and `LinkedList`. It defines signatures for the `add`, `clear`, `iterator`, `remove`, and `size` methods.
interface `Iterator`	`Iterator` defines a simple and consistent interface for iterating over the contents of a collection. Its three methods are `hasNext`, `next`, and `remove`.
interface `List`	`List` is an extension of the `Collection` interface,and provides a means to impose a sequence on the selection. As such, many of its methods take an index parameter: for instance, `add`, `get`, `remove`, and `set`. Classes such as `ArrayList` and `LinkedList` implement `List`.
interface `Map`	The `Map` interface offers an alternative to list-based collections by supporting the idea of associating each object in a collection with a *key* value. Objects are added and retrieved via its `put` and `get` methods. Note that a `Map` does not return an `Iterator`, but its `keySet` method returns a `Set` of the keys, and its `values` method returns a `Collection` of the objects in the map.
interface `Set`	`Set` extends the `Collection` interface with the intention of mandating that a collection contain no duplicate elements. It is worth pointing out that, because it is an interface, `Set` has no actual implication to enforce this restriction. This means that `Set` is actually provided as a marker interface to enable collection implementers to indicate that their classes fulfill this particular restriction.
class `ArrayList`	An implementation of the `List` interface that uses an array to provide efficient direct access via integer indices to the objects it stores. If objects are added or removed from anywhere other than the last position in the list, then following items have to be moved to make space or close the gap. Key methods are `add`, `get`, `iterator`, `remove`, and `size`.
class `Collections`	`Collections` is a collecting point for static methods that are used to manipulate collections. Key methods are `binarySearch`, `fill`, and `sort`.
class `HashMap`	`HashMap` is an implementation of the `Map` interface. Key methods are `get`, `put`, `remove`, and `size`. Iteration over a `HashMap` is usually a two-stage process: obtain the set of keys via its `keySet` method, and then iterate over the keys.
class `HashSet`	`HashSet` is a hash-based implementation of the `Set` interface. It is closer in usage to a `Collection` than to a `HashMap`. Key methods are `add`, `remove`, and `size`.
class `LinkedList`	`LinkedList` is an implementation of the `List` interface that uses an internal linked structure to store objects. Direct access to the ends of the list is efficient, but access to individual objects via an index is less efficient than with an `ArrayList`. On the other hand, adding objects or removing them from within the list requires no shifting of existing objects. Key methods are `add`, `getFirst`, `getLast`, `iterator`, `removeFirst`, `removeLast`, and `size`.
class `Random`	The `Random` class supports generation of pseudo-random values – typically random numbers. The sequence of numbers generated is determined by a *seed value*, which may be passed to a constructor or set via a call to `setSeed`. Two `Random` objects starting from the same seed will return the same sequence of values to identical calls. Key methods are `nextBoolean`, `nextDouble`, `nextInt`, and `setSeed`.
class `Scanner`	The `Scanner` class provides a way to read and parse input. It is often used to read input from the keyboard. Key methods are `next` and `hasNext`.

package `java.io` – Summary of the most important classes and interfaces

interface `Serializable`	The `Serializable` interface is an empty interface requiring no code to be written in an implementing class. Classes implement this interface in order to be able to participate in the serialization process. `Serializable` objects may be written and read as a whole to and from sources of output and input. This makes storage and retrieval of persistent data a relatively simple process in Java. See the `ObjectInputStream` and `ObjectOutputStream` classes for further information.
class `BufferedReader`	`BufferedReader` is a class that provides buffered character-based access to a source of input. Buffered input is often more efficient than unbuffered, particularly if the source of input is in the external file system. Because it buffers input, it is able to offer a `readLine` method that is not available in most other input classes. Key methods are `close`, `read`, and `readLine`.
class `BufferedWriter`	`BufferedWriter` is a class that provides buffered character-based output. Buffered output is often more efficient than unbuffered, particularly if the destination of the output is in the external file system. Key methods are `close`, `flush`, and `write`.
class `File`	The `File` class provides an object representation for files and folders (directories) in an external file system. Methods exist to indicate whether a file is readable and/or writeable, and whether it is a file or a folder. A `File` object can be created for a non-existent file, which may be a first step in creating a physical file on the file system. Key methods are `canRead`, `canWrite`, `createNewFile`, `createTempFile`, `getName`, `getParent`, `getPath`, `isDirectory`, `isFile`, and `listFiles`.
class `FileReader`	The `FileReader` class is used to open an external file ready for reading its contents as characters. A `FileReader` object is often passed to the constructor of another reader class (such as a `BufferedReader`) rather than being used directly. Key methods are `close` and `read`.
class `FileWriter`	The `FileWriter` class is used to open an external file ready for writing character-based data. Pairs of constructors determine whether an existing file will be appended or its existing contents discarded. A `FileWriter` object is often passed to the constructor of another `Writer` class (such as a `BufferedWriter`) rather than being used directly. Key methods are `close`, `flush`, and `write`.
class `IOException`	`IOException` is a checked exception class that is at the root of the exception hierarchy of most input/output exceptions.

package `java.net` – Summary of the most important classes	
class URL	The URL class represents a Uniform Resource Locator: in other words, it provides a way to describe the location of something on the Internet. In fact, it can also be used to describe the location of something on a local file system. We have included it here because classes from the `java.io` and `javax.swing` packages often use URL objects. Key methods are `getContent`, `getFile`, `getHost`, `getPath`, and `openStream`.

Other important packages

Other important packages are

```
java.awt
java.awt.event
javax.swing
javax.swing.event
```

These are used extensively when writing graphical user interfaces (GUIs), and they contain many useful classes that a GUI programmer should become familiar with.

Additional Resources

US Edition

David J. Barnes & Michael Kölling

Objects First with Java

A Practical Introduction using BlueJ

Sixth Edition, Pearson, 2016

ISBN (US edition): 978-013-447736-7
ISBN (Global Edition): 978-1-292-15904-1

Global Edition

New: Sixth Edition out now

book features

• objects-first approach • project driven • spiral approach
• thorough treatment of object-oriented principles • includes new Java 8 features: streams, lambdas

Book Information

New in the 6th edition

Description

Chapter Sequence

Evaluation

Errata

Resources

Book Projects (zip file, 37Mb)

Code Style Guide

Resources mentioned in the book

Teacher Resources (incl. slides)

The Blueroom (teacher community)

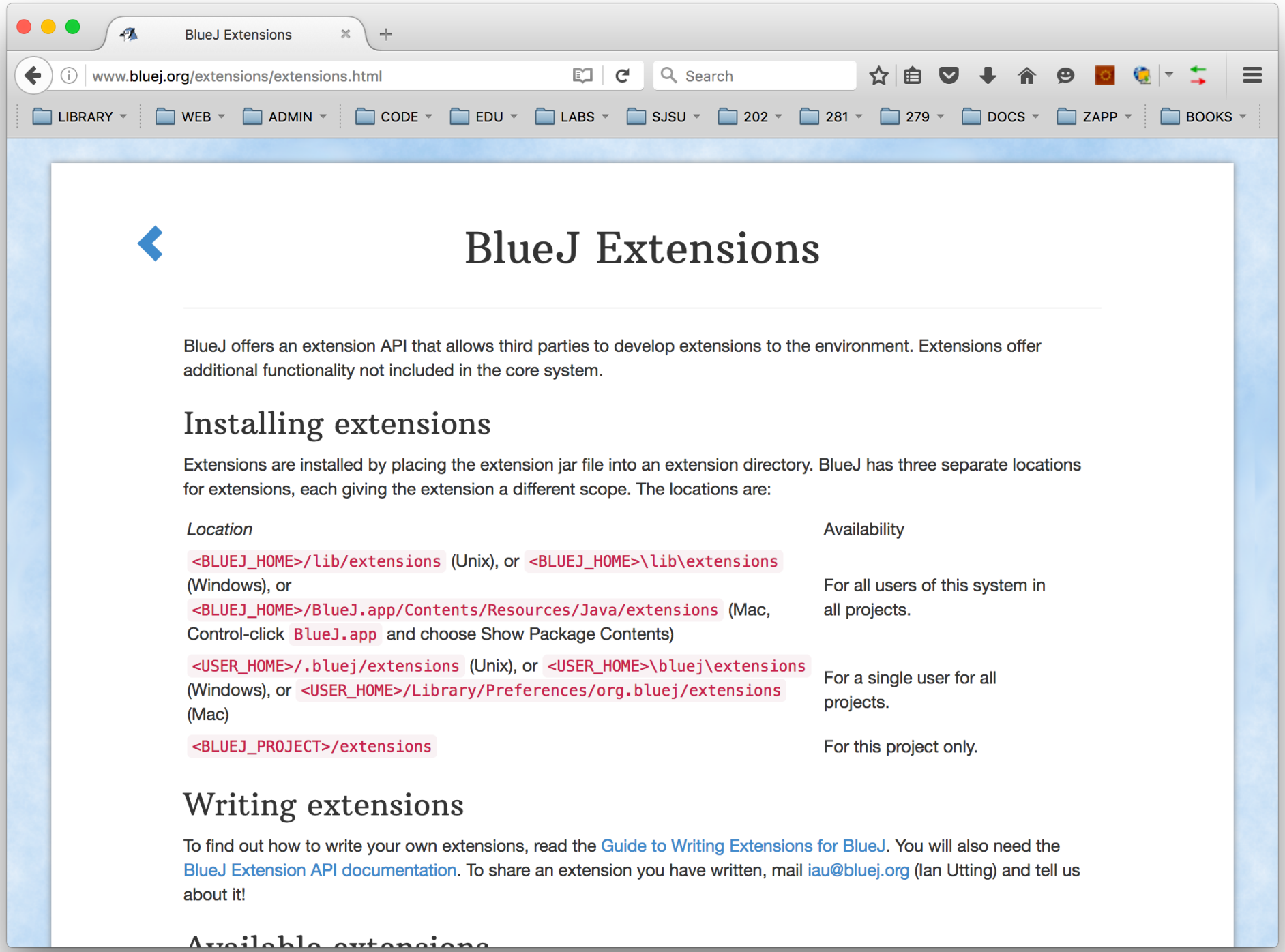
Video Notes
(made for 5th edition, but usable with 6th edition as well)

Additional Information

Translations

Previous editions:
5th / 4th / 3rd / 2nd / 1st

BlueJ



BlueJ Extensions

BlueJ offers an extension API that allows third parties to develop extensions to the environment. Extensions offer additional functionality not included in the core system.

Installing extensions

Extensions are installed by placing the extension jar file into an extension directory. BlueJ has three separate locations for extensions, each giving the extension a different scope. The locations are:

Location	Availability
`<BLUEJ_HOME>/lib/extensions` (Unix), or `<BLUEJ_HOME>\lib\extensions` (Windows), or `<BLUEJ_HOME>/BlueJ.app/Contents/Resources/Java/extensions` (Mac, Control-click `BlueJ.app` and choose Show Package Contents)	For all users of this system in all projects.
`<USER_HOME>/.bluej/extensions` (Unix), or `<USER_HOME>\bluej\extensions` (Windows), or `<USER_HOME>/Library/Preferences/org.bluej/extensions` (Mac)	For a single user for all projects.
`<BLUEJ_PROJECT>/extensions`	For this project only.

Writing extensions

To find out how to write your own extensions, read the Guide to Writing Extensions for BlueJ. You will also need the BlueJ Extension API documentation. To share an extension you have written, mail iau@bluej.org (Ian Utting) and tell us about it!

Available extensions