

Objects and Classes

Control Flow in Java

Object Interaction



Objects and Classes

The foundation of Object Oriented
Programming



Fundamental Concepts

- object
- class
- method
- parameter
- data type



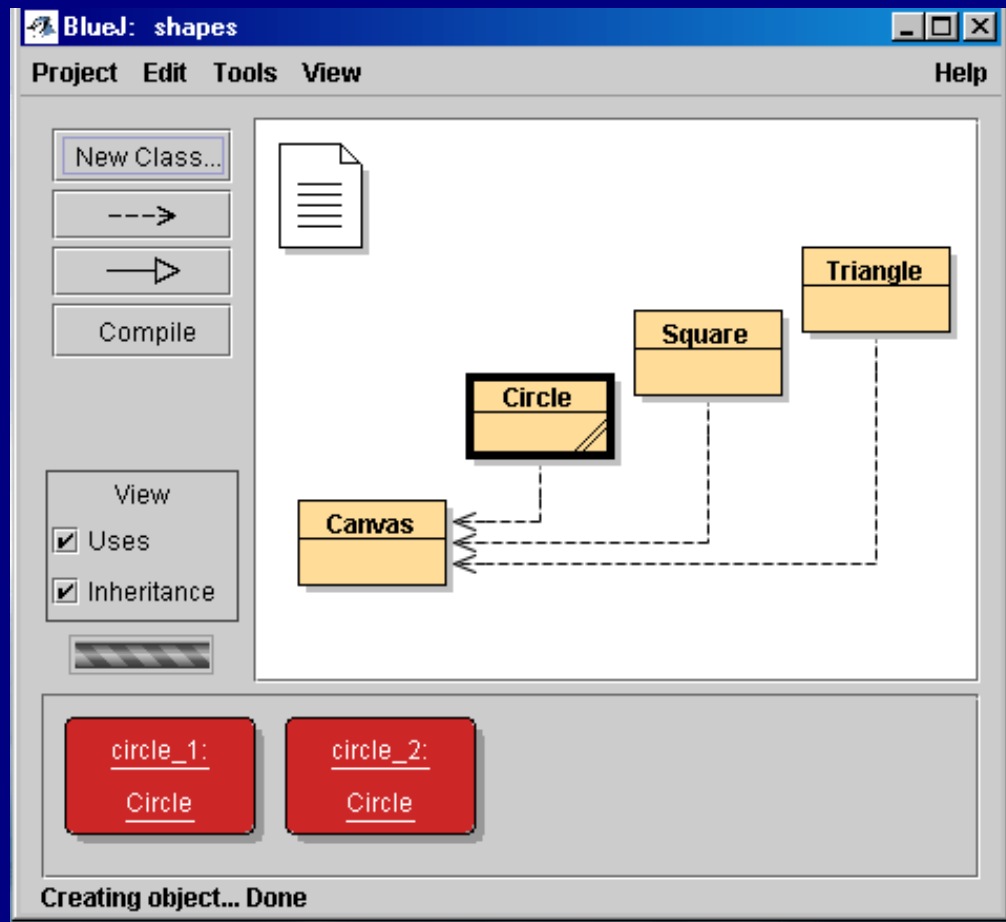
Objects and Classes

- Objects
 - represent ‘things’ from the real world, or from some problem domain (example: “the red car down there in the car park”)
- Classes
 - represent all objects of a kind (example: “car”)

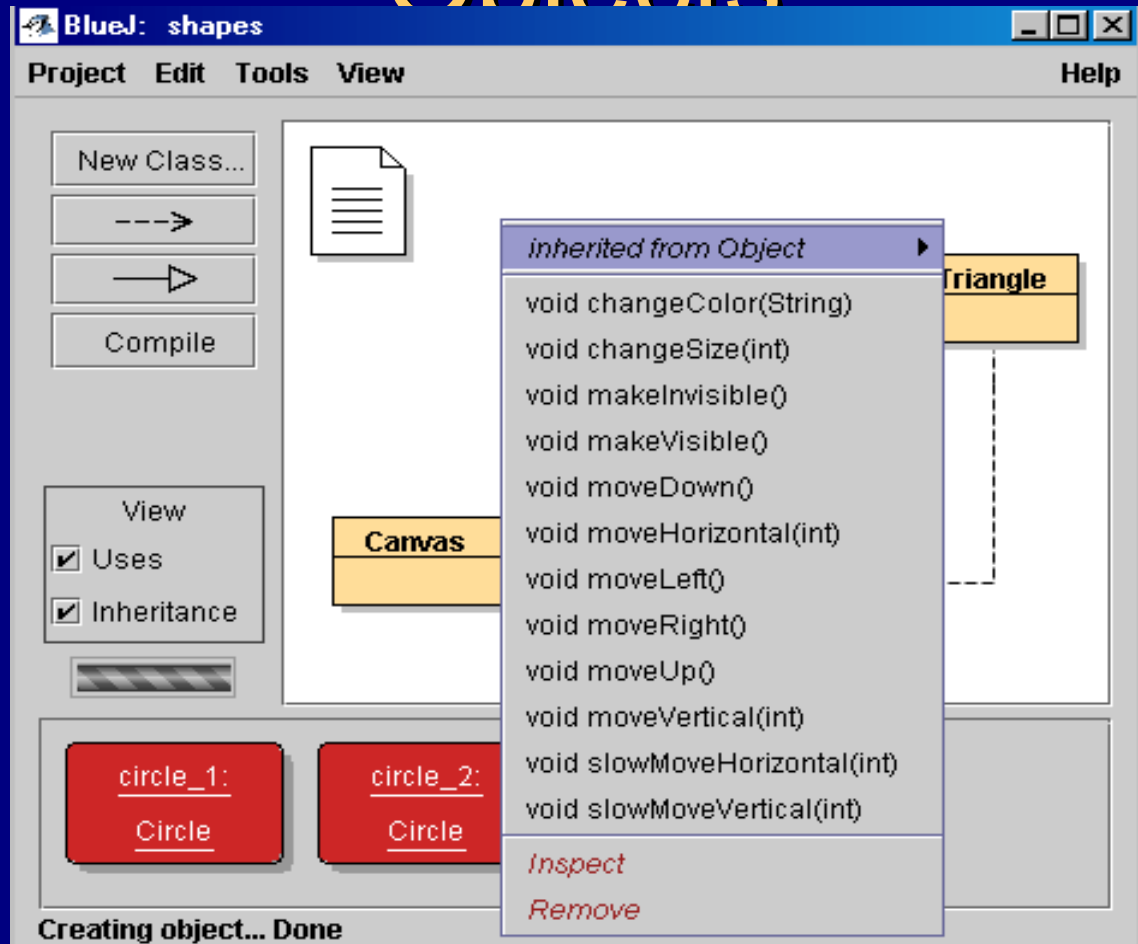
Objects represent individual instantiations of the class. Object are instantiated.



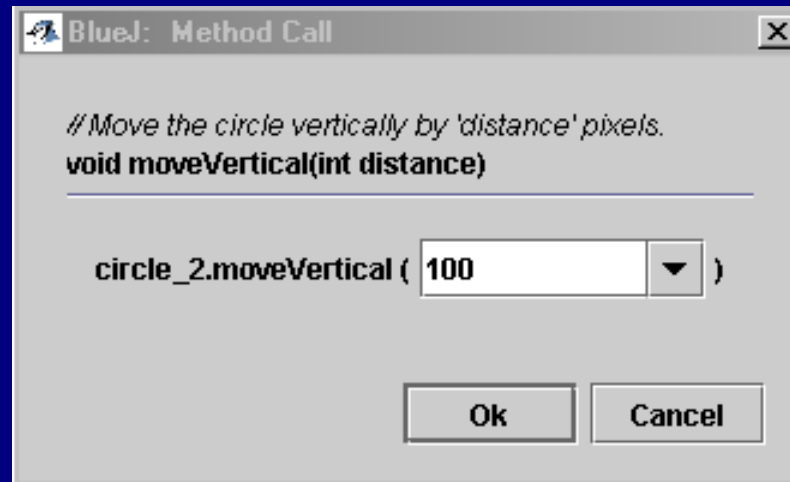
Objects and Classes in BlueJ



Things we can do with Objects



Things we can do with Objects



Methods and Parameters

- Objects/classes have operations which can be invoked. They are called methods
- void moveHorizontal(int distance) is called the signature of the methods
- The collection of methods of a class is referred to as the interface of that class
- methods may have parameters to pass additional information needed to execute
- Methods are called or invoked



Data Types

- Parameters have types. A type defines what kinds of values a parameter can take.
- Defining a class defines a type
- In Java, everything has a type.
- Java is strongly typed language
- Examples of types: int, String, Circle, ...

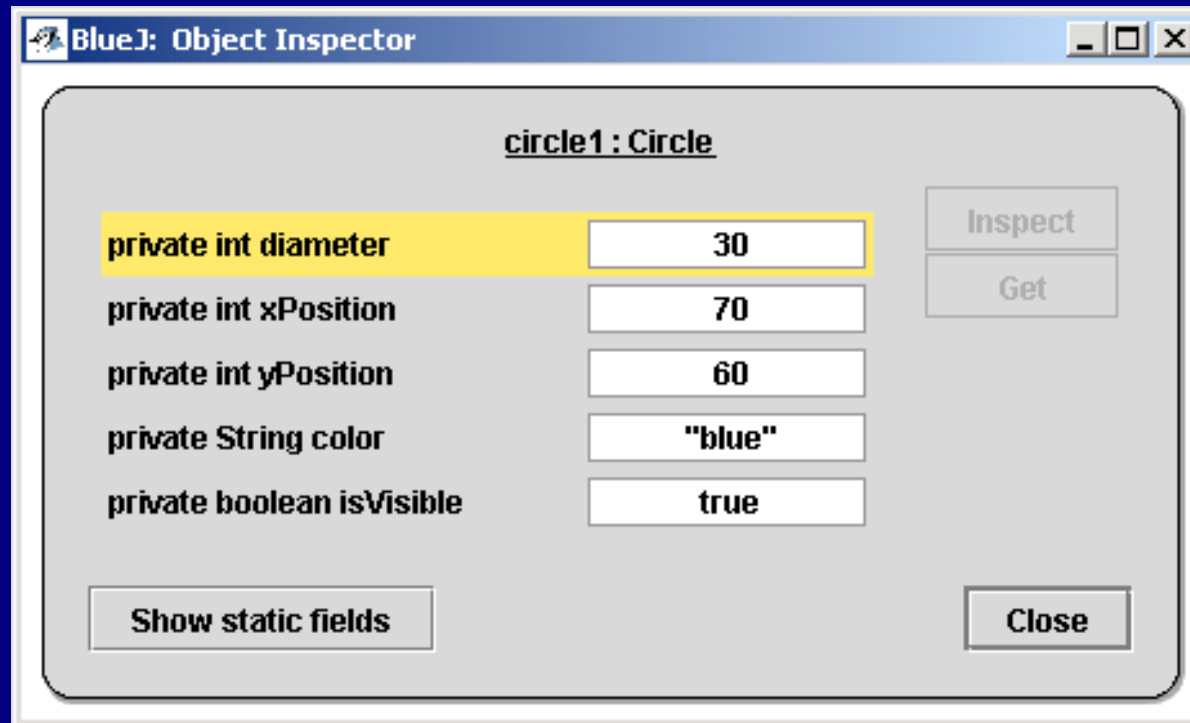


Other Observations

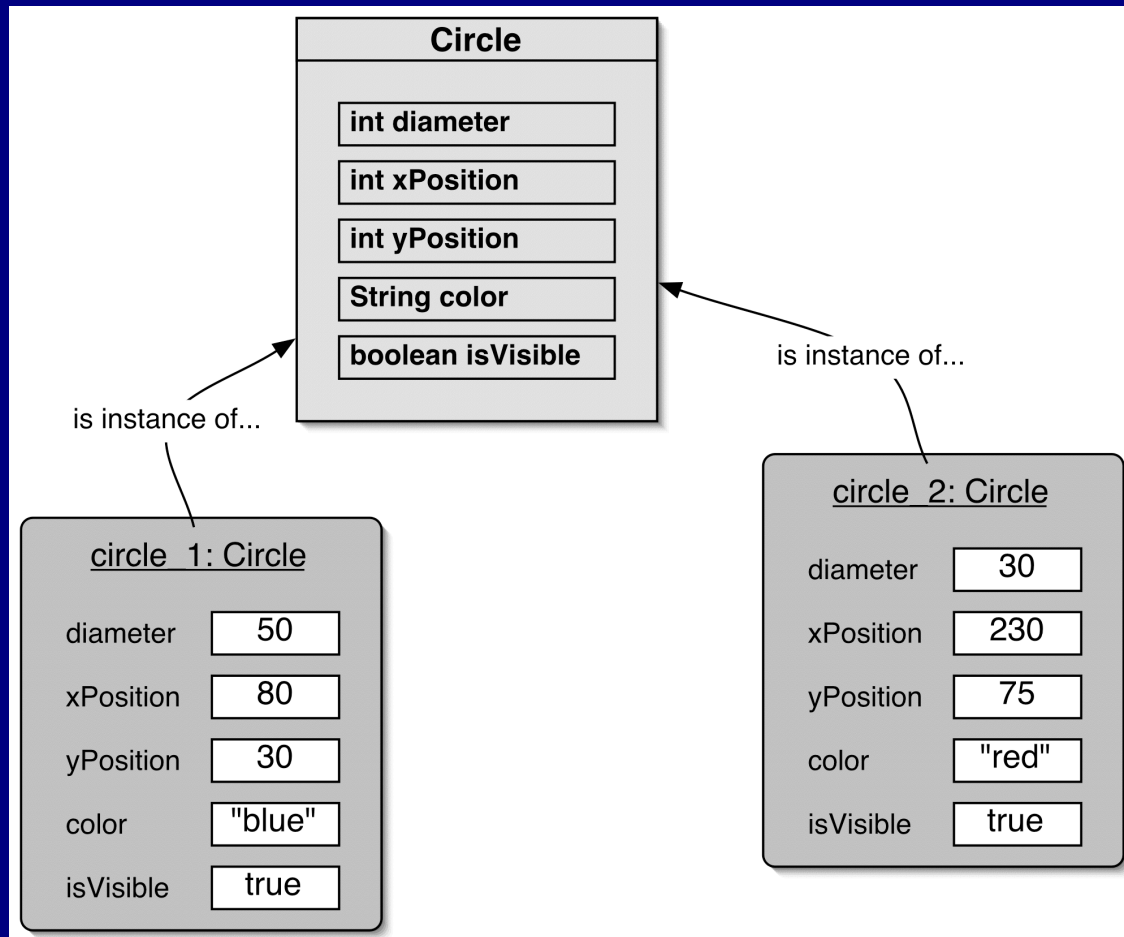
- many *instances* can be created from a single class
- an object has *attributes*: values stored in *fields*.
- the class defines what fields an object has, but each object stores its own set of values.
- These set of values is called the *state* of the object.



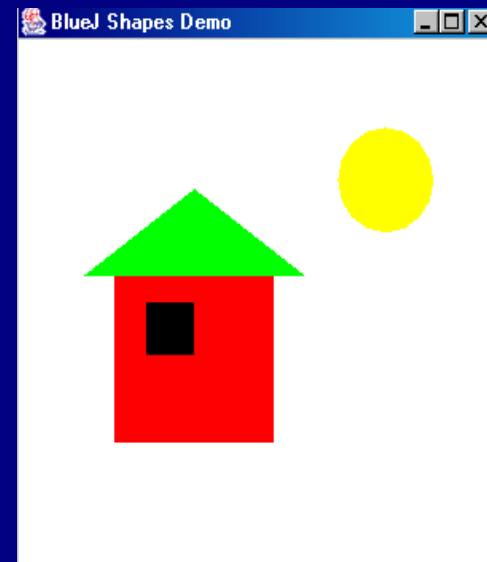
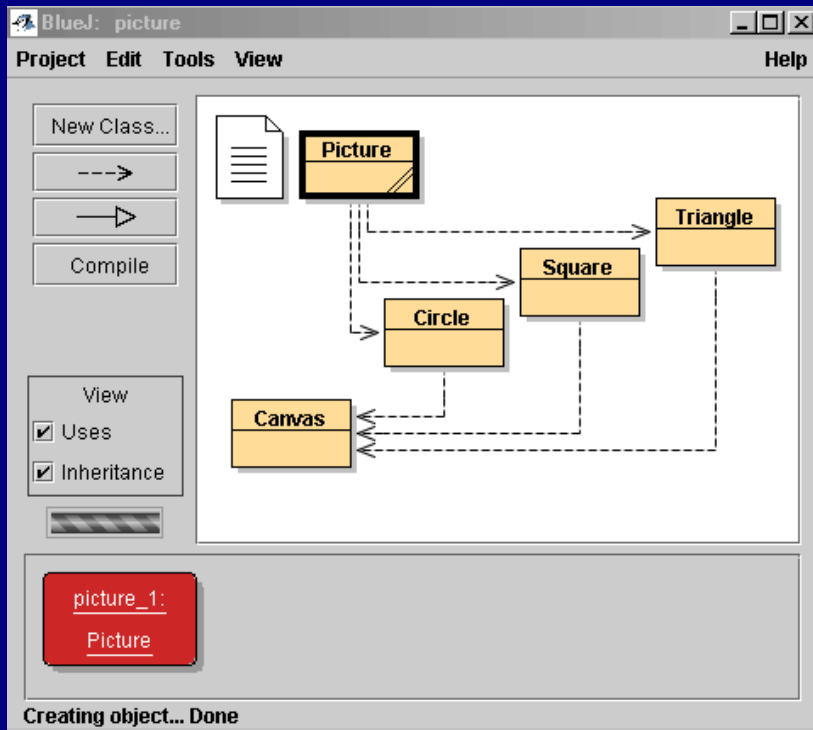
State



Two Circle Objects



Object Interaction



Source Code

- Each class has source code (Java code) associated with it that defines its details (fields and methods).
- In other words, it determines the structure and the behavior of each of its instance.
- This source code is compiled and interpreted by Java.



Return Values

- Methods may return a result via a return value.
- Example: `String getName()`
 - This method returns a String.
- Example: `void changeName()`
 - Void indicates that this method does not return anything



Developing Java Programs

- To learn to develop Java programs, one needs to learn how to write class definitions, including fields and methods, and how to put these classes together as well
- During the rest of this unit we will deal with these issues in more detail



Coding Conventions

- Classes: Uppercase to start, merge words, consecutive words uppercase, nouns
 - E.g. Car, Number, BankAccount
- Objects: Lowercase to start, merge words, consecutive words uppercase, nouns
 - E.g. myBlueCar, Rational
- Methods: Lowercase to start, merge words, consecutive words uppercase, verbs
 - E.g. moveLocation, deposit



Terms

- Object
- Instance
- State
- Class
- Method
- Return Value
- Signature
- Parameter
- Type
- Source Code
- Compiler
- Virtual Machine
- Method Calling



Understanding class definitions

Looking inside classes



Main concepts to be covered

- fields
- constructors
- methods
- parameters
- assignment statements
- conditional statements

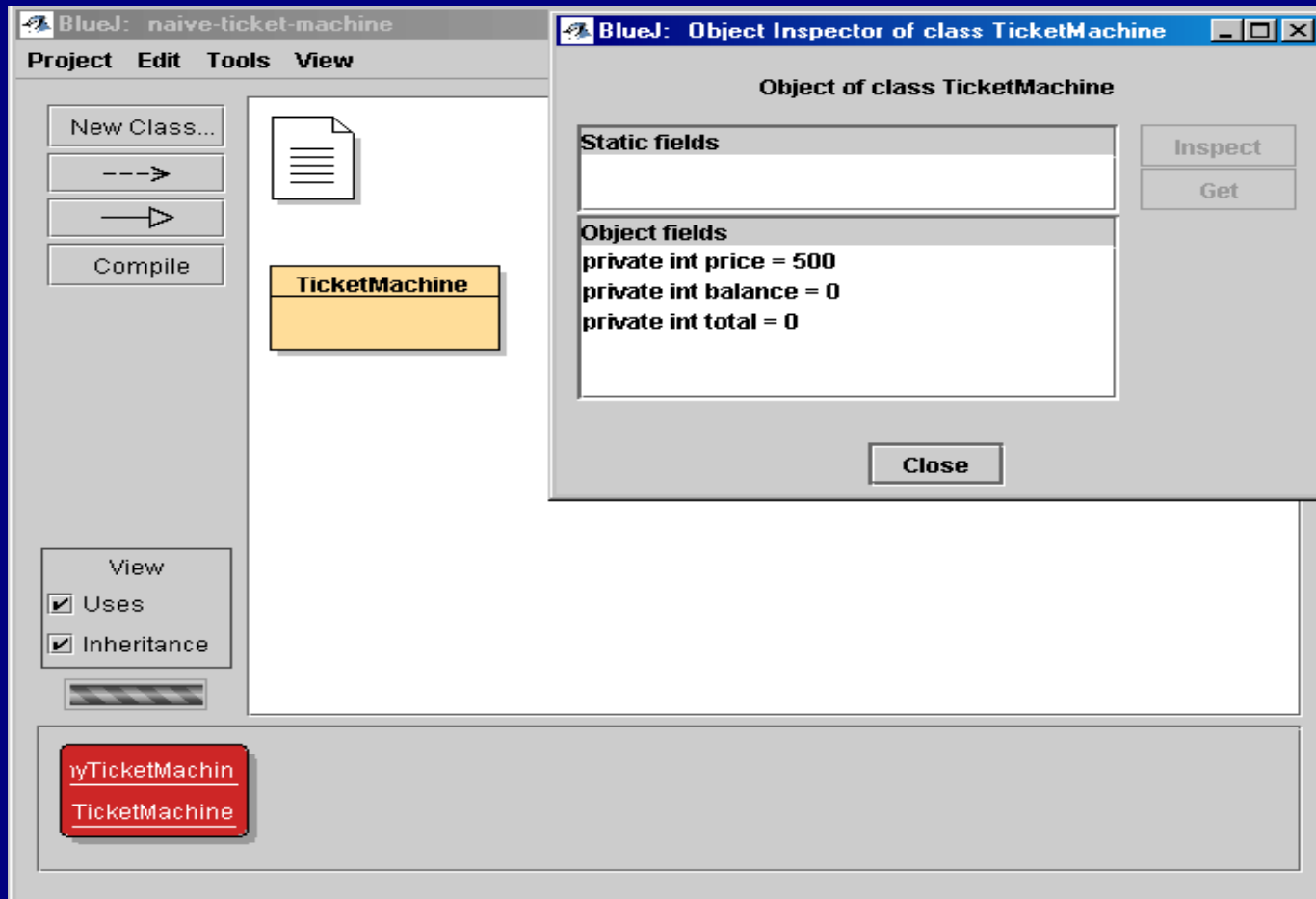


Ticket machines – an external view

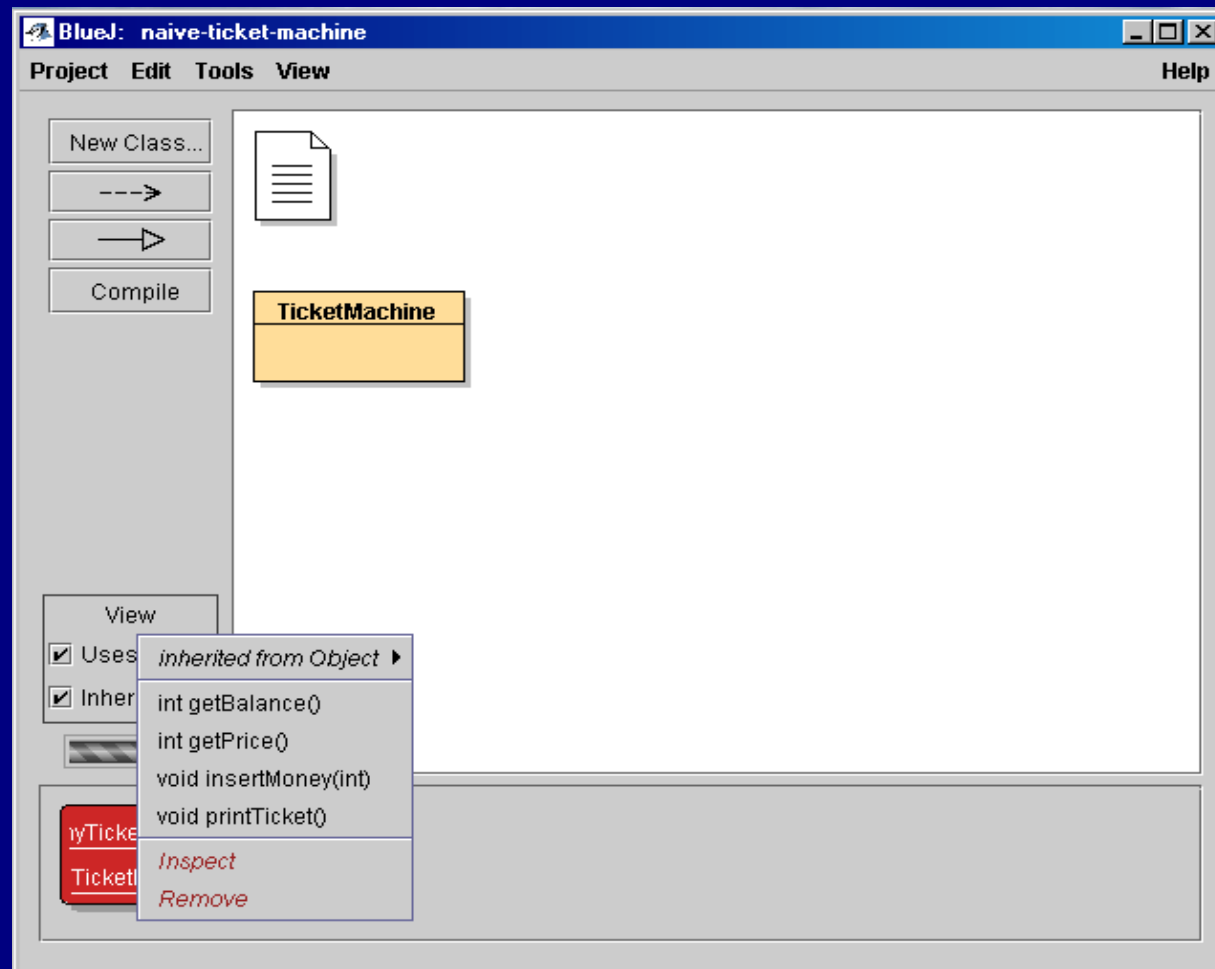
- Exploring the behaviour of a typical ticket machine.
 - Use the *naive-ticket-machine* project.
 - Machines supply tickets of a fixed price.
 - How is that price determined?
 - How is ‘money’ entered into a machine?
 - How does a machine keep track of the money that is entered?
 - How is a ticket provided?



Resulting Fields



Resulting Methods

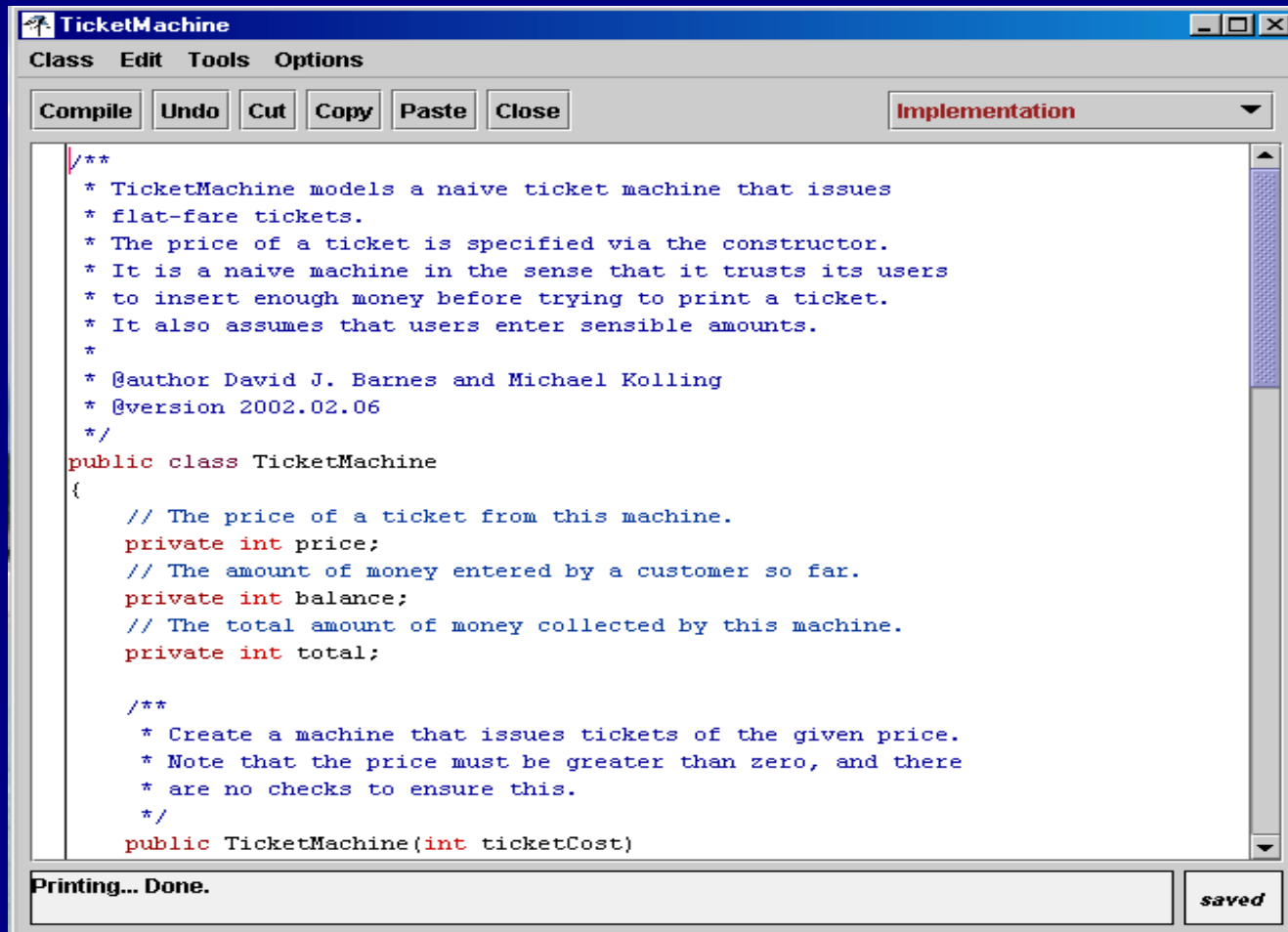


Ticket machines – an internal view

- Interacting with an object gives us clues about its behavior.
- Looking inside allows us to determine how that behavior is provided or implemented.
 - Looking at the source code
- All Java classes have a similar-looking internal view.



The Source Code



The screenshot shows a Java IDE window titled "TicketMachine". The menu bar includes "Class", "Edit", "Tools", and "Options". Below the menu bar is a toolbar with buttons for "Compile", "Undo", "Cut", "Copy", "Paste", and "Close". To the right of the toolbar is a dropdown menu currently set to "Implementation". The main text area contains the following Java code:

```
/**
 * TicketMachine models a naive ticket machine that issues
 * flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * It is a naive machine in the sense that it trusts its users
 * to insert enough money before trying to print a ticket.
 * It also assumes that users enter sensible amounts.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2002.02.06
 */
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.
     * Note that the price must be greater than zero, and there
     * are no checks to ensure this.
     */
    public TicketMachine(int ticketCost)
```

At the bottom of the window, there is a status bar that says "Printing... Done." and a button labeled "saved".



Basic class structure

```
public class TicketMachine  
{  
    Inner part of the class omitted.  
}
```

The outer wrapper
of TicketMachine

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

The contents of a
class



Comments/Documentation

- Comments make source code easier to read for humans. No effect on the functionality.
- Three sorts:
 - `//` comment: single-line comments
 - `/*` comments `*/`: multiple-lines – more detail
 - `/**` `*/`: similar to previous, but used when documentation software is used.



Fields

- Fields store values for an object.
- They are also known as instance variables.
- Use the *Inspect* option to view an object's fields.
- Fields define the state of an object.

```
public class TicketMachine  
{
```

```
    private int price;  
    private int balance;  
    private int total;
```

Constructor and methods omitted.

```
}
```

visibility modifier type variable name

↓ ↓ ↓

```
private int price;
```

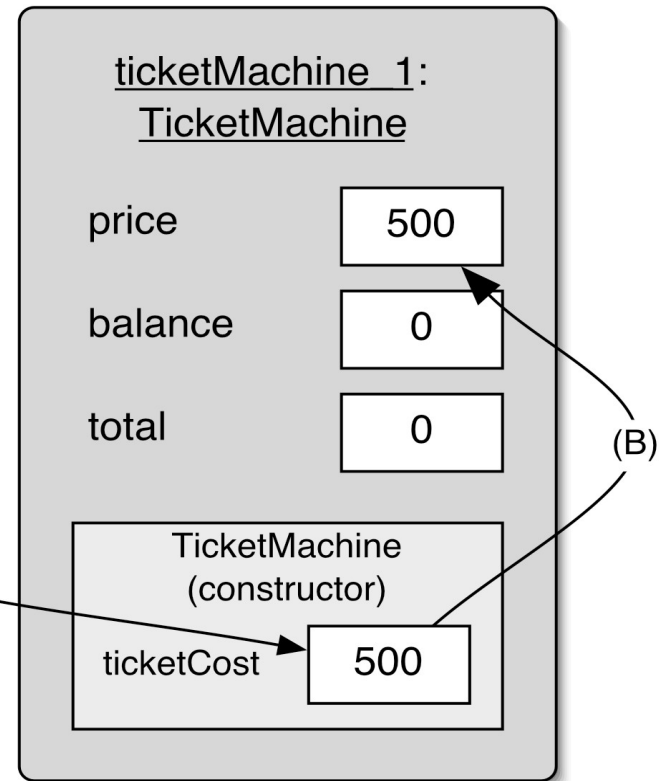
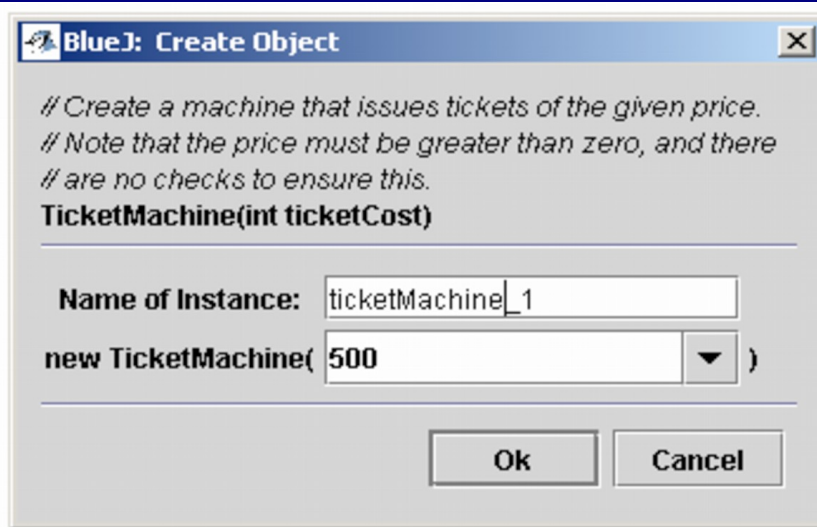


Constructors

- Constructors initialize an object.
 - Then assign the necessary memory to the created object
 - They have the same name as their class.
 - They store initial values into the fields.
 - They often receive external parameter values for this.
- ```
public TicketMachine(int ticketCost)
{
 price = ticketCost;
 balance = 0;
 total = 0;
}
```



# Passing data via parameters



# Parameters

- Parameter names inside a constructor or method are referred to as **Formal Parameters.**
- Parameter values provided from the outside are referred to as **Actual Parameters.**
- In the example: ticketCost is a formal parameter and 500 is an actual parameter.



# Space

- The ticketCost box in the object representation is only created when the constructor is executed.
- Extra temporarily storage is provided to store a value for ticketCost. This is called the constructor space or method space.
- Values can only be used during the execution.





# Scope and Lifetime

- The scope of a variable/parameter defines the section of the code from where it can be accessed.
- For instance variables this is the entire class.
- For parameters, this is the constructor or method that declares it.
- Trick: find the enclosing {}, this is the scope
- The lifetime of a variable/parameter describes how long the variable continues to exist before it is destroyed.



# Assignment

- Values are stored into fields (and other variables) via assignment statements:
  - *variable = expression;*
  - `price = ticketCost;`
- Both sides of the assignment should have the same type, e.g. int, double, String, ...
- A variable stores a single value, so any previous value is lost.

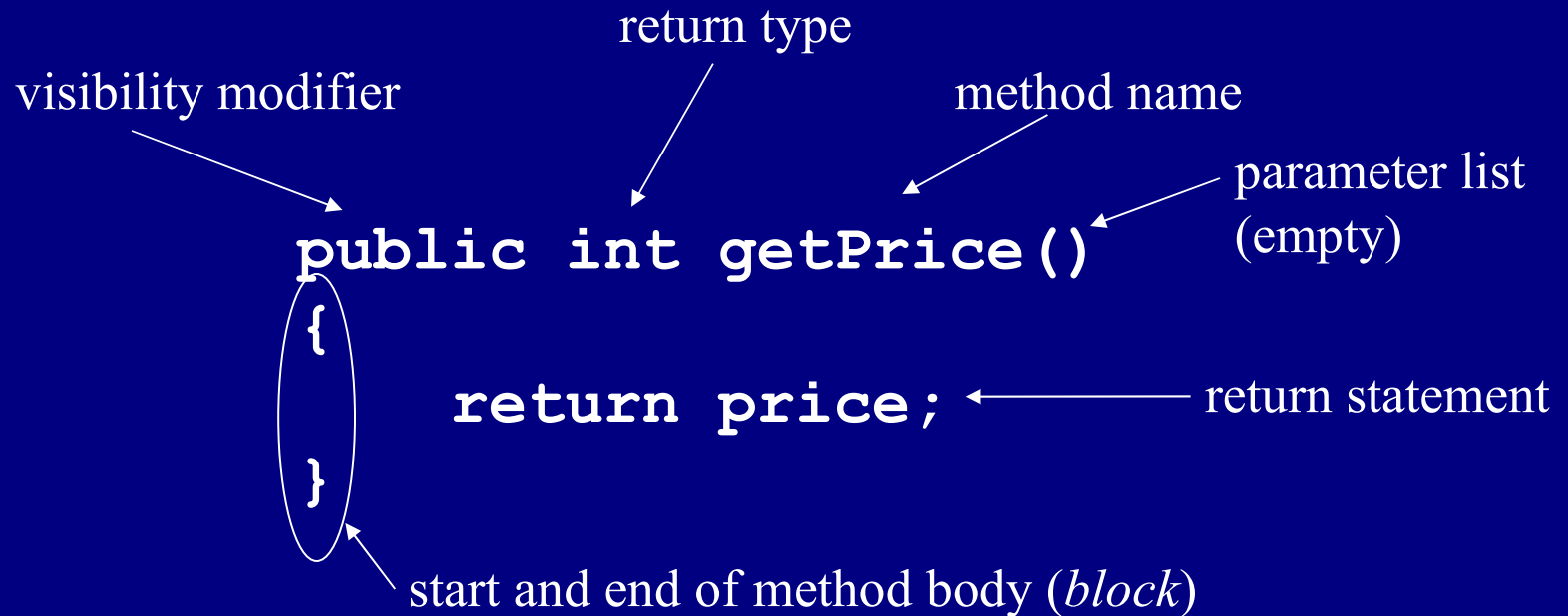


# Accessor methods

- Methods implement the behavior of objects.
- Accessors provide information about an object.
- Methods have a structure consisting of a header and a body.
- The header defines the method's *signature*.  
`public int getPrice()`
- The body encloses the method's statements.



# Accessor methods



# Mutator methods

- Have a similar method structure: header and body.
- Used to *mutate* (i.e., change) an object's state.
- Achieved through changing the value of one or more fields.
  - Typically contain assignment statements.
  - Typically receive parameters.



# Mutator methods

visibility modifier    return type (`void`)    method name    parameter

```
public void insertMoney(int amount)
{
 balance += amount;
}
```

assignment statement

field being changed



# Abstract Data Types

- Classes define types
  - Can be used as parameter, field and return types
- The internal is hidden from the user
  - No direct access to fields (unless special reason)
  - Access to state via accessor and mutator methods
- User does not need to know how the class is implemented to use/instantiate it
- The usage of a class is defined by its methods



# Printing from methods

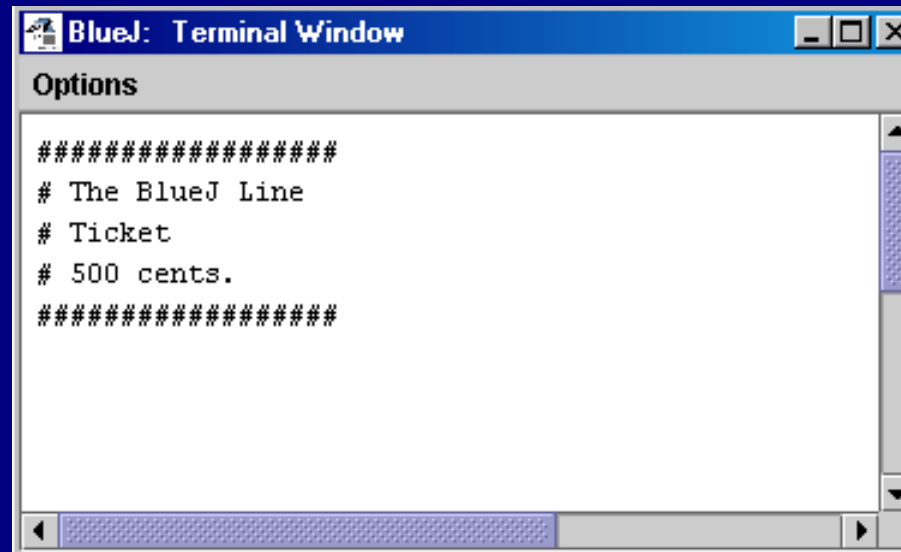
```
public void printTicket()
{
 // Simulate the printing of a ticket.
 System.out.println("#####");
 System.out.println("# The BlueJ Line");
 System.out.println("# Ticket");
 System.out.println("# " + price + " cents.");
 System.out.println("#####");
 System.out.println();

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





# Output

A screenshot of a BlueJ Terminal Window. The window has a title bar that says "BlueJ: Terminal Window" with standard minimize, maximize, and close buttons. Below the title bar is a tab labeled "Options". The main area of the window is a text editor showing the following output:

```

The BlueJ Line
Ticket
500 cents.
#####
```

The text is in a monospaced font. There are scrollbars on the right and bottom of the text area.

# Reflecting on the ticket machines

- Their behavior is inadequate in several ways:
  - No checks on the amounts entered.
  - No refunds.
  - No checks for a sensible initialization.
- How can we do better?
  - We need more sophisticated behavior.



# Making choices

```
public void insertMoney(int amount)
{
 if(amount > 0) {
 balance += amount;
 }
 else {
 System.out.println("Use a positive amount: " +
 amount);
 }
}
```



# Making choices

*'if' keyword*

*boolean condition to be tested - gives a *true* or *false* result*

*actions if condition is true*

```
if(perform some test) {
 Do the statements here if the test gave a true result
}
else {
 Do the statements here if the test gave a false result
}
```

*'else' keyword*

*actions if condition is false*



# Boolean Tests

- `==` : equality
- `>` : greater than
- `<` : less than
- `<=` : less or equal than
- `>=` : greater or equal than
- `!=` : not equal

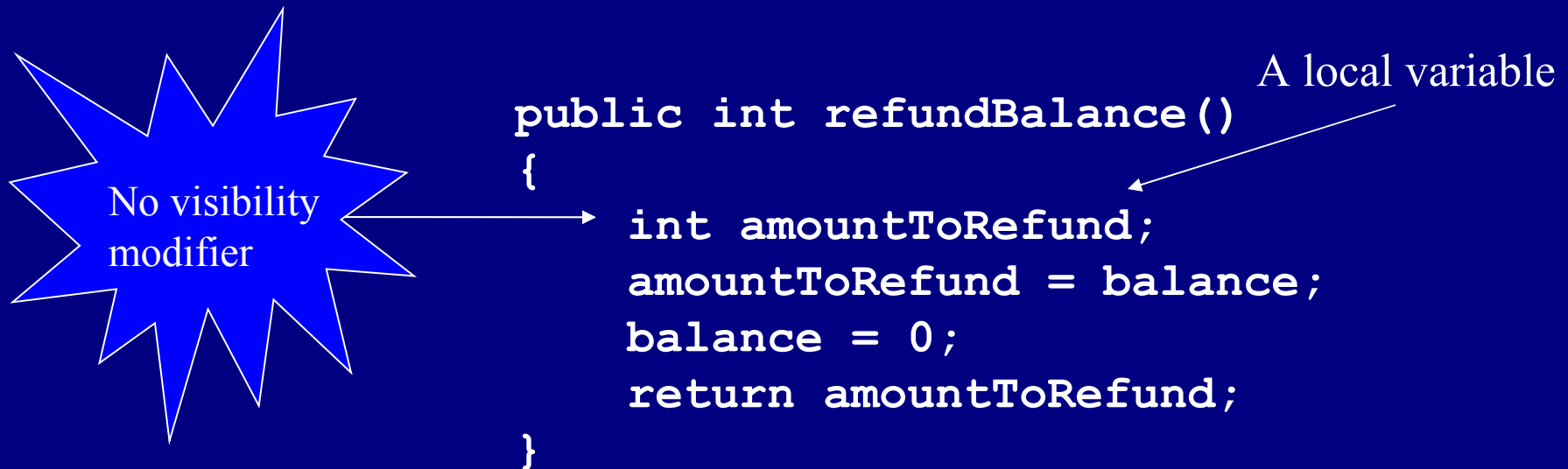


# Local variables

- Fields are one sort of variable.
  - They store values through the life of an object.
  - They are accessible throughout the class.
- Methods can include shorter-lived variables.
  - They exist only as long as the method is being executed.
  - They are only accessible from within the method.



# Local variables



# Review

- Class bodies contain fields, constructors and methods.
- Fields store values that determine an object's state.
- Constructors initialize objects.
- Methods implement the behavior of objects.
- Constructors are methods which do not return anything.





# Review

- Fields, parameters and local variables are all variables.
- Fields persist for the lifetime of an object.
- Parameters are used to receive values into a constructor or method.
- Local variables are used for short-lived temporary storage.



# Review

- Objects can make decisions via conditional (if) statements.
- A true or false test allows one of two alternative courses of actions to be taken.



# Coding Convention

- If statement
  - Always use { , even if there is only one statement
  - In case there is an else statement, start on a new line and use {
- Indentation
  - Always indent your code, even if your text editor does not do it automatically
- Document your code, the sooner the better.



# Terms

- Instance variables
- Local variables
- Parameters
- Formal Parameters
- Actual Parameters
- Scope
- Lifetime
- Constructors
- Methods
- If-statement
- Assignment
- =
- +=
- <=, >=, <, >, !=, ==



# Object interaction

Creating cooperating objects



# Main concepts to be covered

- Abstraction
- Modularization
- Class and Object Diagrams
- Call-by-reference and Call-by-value
- Overloading
- Internal and External method calls
- this keyword
- Debugging



# A digital clock

A digital clock display showing the time 11:03. The digits are black and bold, set against a white rectangular background with a thin black border. The entire display is centered on a dark blue background.

11:03



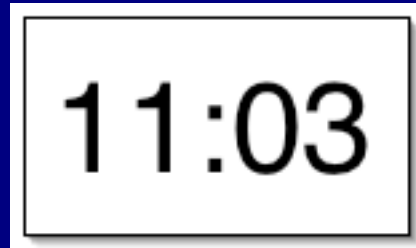
# Abstraction and modularization

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



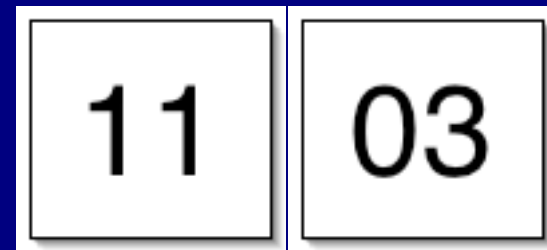


# Modularizing the clock display



One four-digit display?

Or two two-digit displays?



# Implementation: NumberDisplay

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

 Constructor and
methods omitted.
}
```



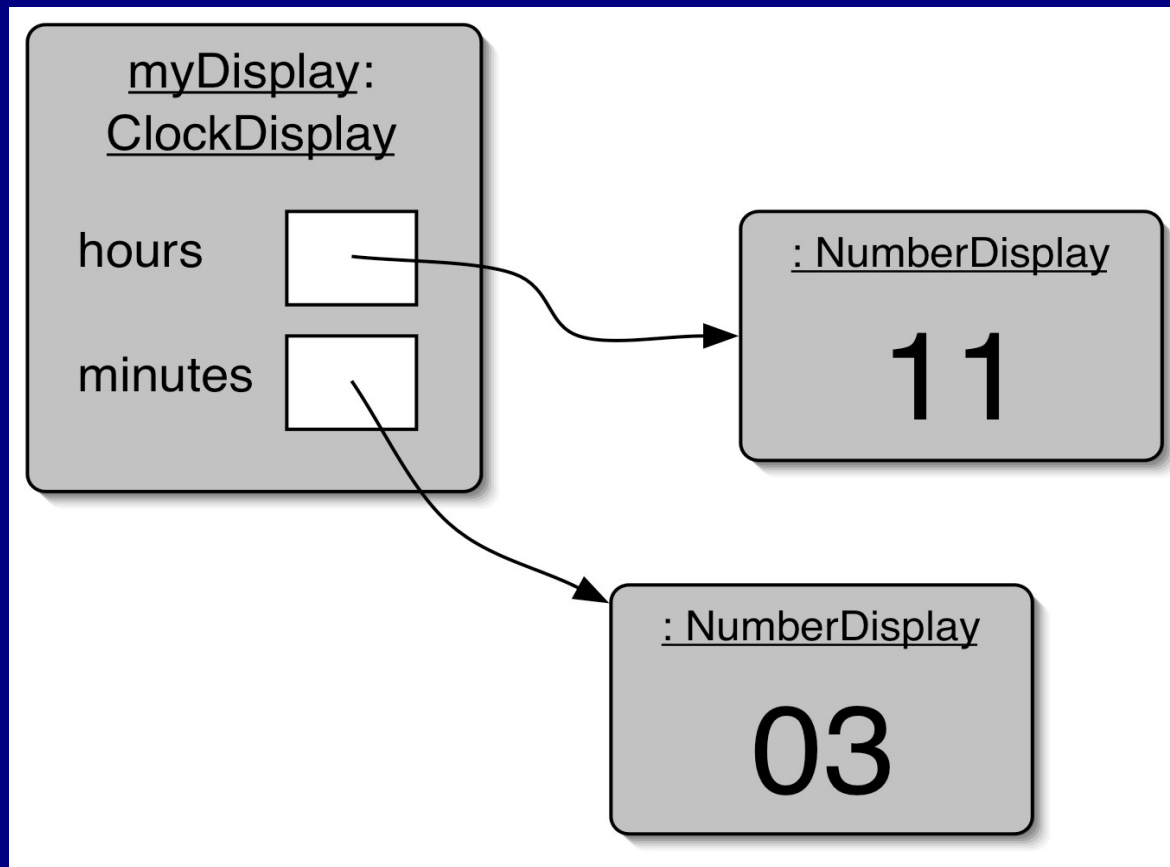
# Implementation ClockDisplay

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

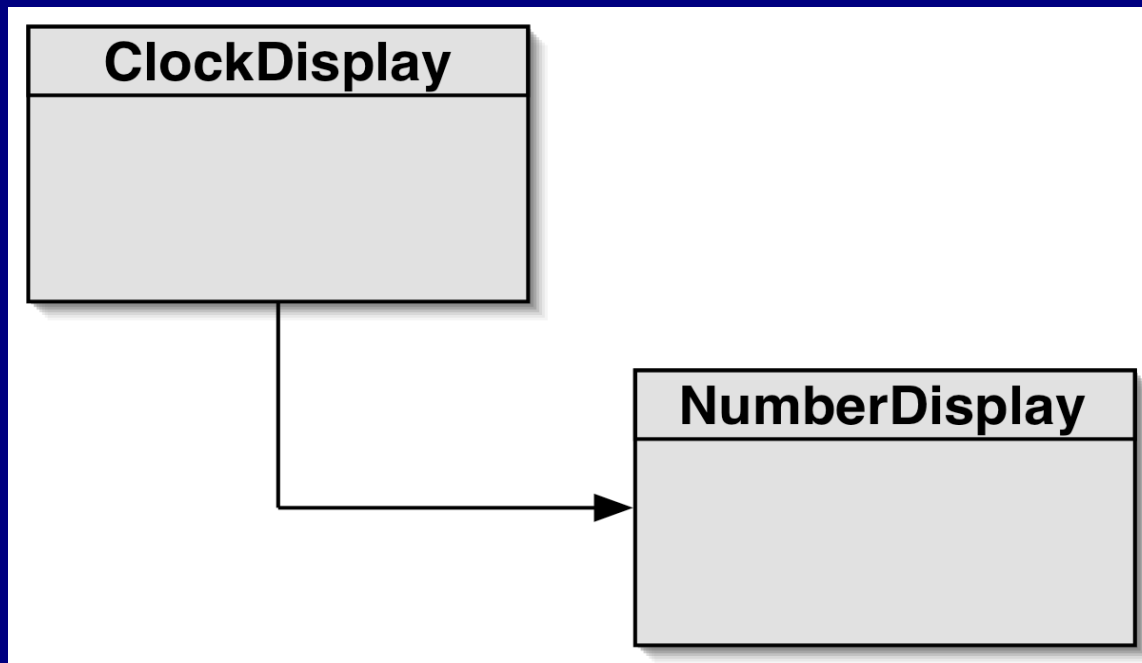
 Constructor and
methods omitted.
}
```



# Object diagram



# Class diagram

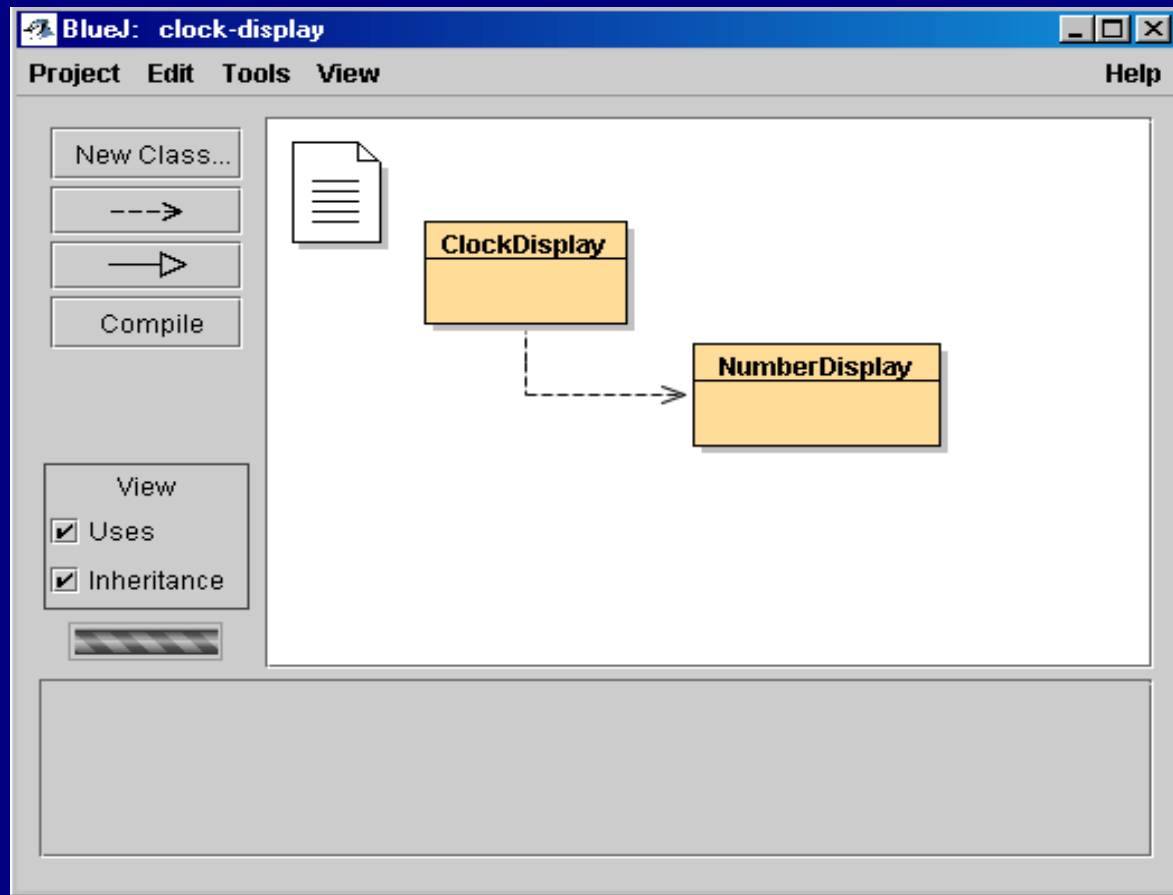


# Diagrams

- Class Diagrams
  - Shows the classes of an application and the relationships between them
  - Gives information about the source code
  - Static view of the program
- Object Diagrams
  - Shows objects and their relationships at one moment in time during the execution of the program
  - Dynamic view of the program



# BlueJ and Diagrams



# Primitive types vs. object types

- Java defines two very different kinds of type: primitive types and object types.
- Primitive types are predefined by Java.
- Object types originate from classes.
- Variables and parameters store references to objects.
- The primitive types are non-object types.

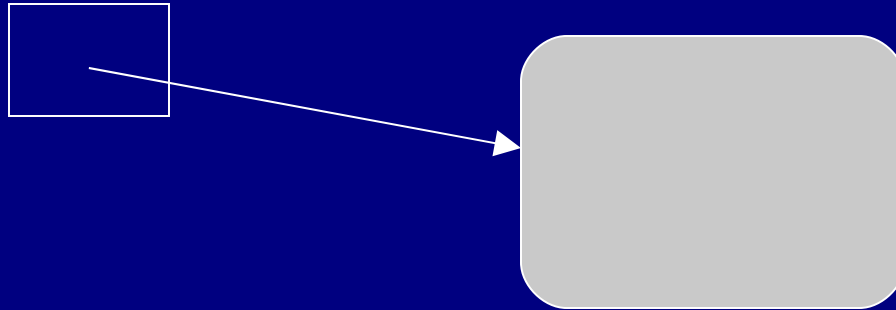




# Primitive types vs. object types

`SomeObject obj;`

object type



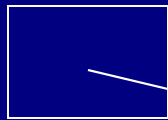
`int i;`

primitive type

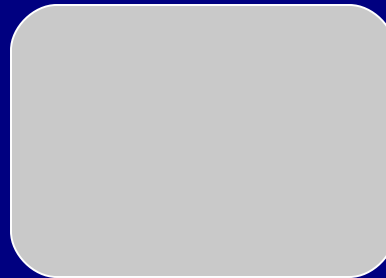
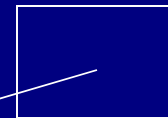


# Primitive types vs. object types

`SomeObject a;`



`SomeObject b;`



---

`b = a;`

`int a;`



`int b;`



# Call-by-reference and Call-by-value

- There are two ways of passing arguments to methods in many programming languages: call-by-value and call-by-reference.
- Call-by-value: A copy of the actual parameter is passed to the formal parameter of the called method. Any change made to the formal parameter will have no effect on the actual parameter.



# Call-by-reference and Call-by-value

- Call-by-reference: the caller gives the called method the ability to directly access to the caller's data and to modify that data if the called method so chooses.
- Java uses call-by-value



# Source code: NumberDisplay

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

 public NumberDisplay(int rollOverLimit)
 {
 limit = rollOverLimit;
 value = 0;
 }
}
```



# Source code: NumberDisplay

```
public int getValue()
{
 return value;
}
```

```
public void setValue(int replacementValue)
{
 if((replacementValue >= 0) &&
 (replacementValue < limit))
 value = replacementValue;
}
```



# Logical Operators

- `&&` : and, operands are tested, left to right, until conclusion can be reached
- `||` : or, operands are tested, left to right, until conclusion can be reached
- `!` : not
- `&` : and, both operands are tested
- `|` : or, both operands are tested



# Source code: NumberDisplay

```
public String getDisplayValue()
{
 if(value < 10)
 return "0" + value;
 else
 return "" + value;
}

public void increment()
{
 value = (value + 1) % limit;
}
}
```





# String Concatenation

- Addition:
  - $12 + 24$
- String Concatenation:
  - “Java” + “with BlueJ” -> “Javawith BlueJ”
  - “answer: ” + 42 -> “answer: 42”



# String toString() method

- String toString() method: Java provides a way of transforming every Object into a String. To tailor this to your own preference write a method toString() returning a String representation of your class/object.

```
public String toString()
{
 return "value: " + value + " with limit " + limit;
}
```



# The Modulo Operator

- `%` : the modulo operator calculates the remainder of an integer division
  - `27 % 4 -> 3`
- Division in Java: if both arguments are integers, division will result in an integer.
  - `double res = 5 / 2 -> res = 2`
  - `double res = 5 / (2.0) or 5 / (2 * 1.0)`  
`-> res = 2.5`



# Objects creating objects

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

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



# Objects creating objects

## 1. new ClassName(parameter-list)

- It creates a new object of the named class
  - here NumberDisplay
  - this involves creating sufficient memory to store the values of primitive instance variables and references to object instance variables.

## 2. It executes the constructor of that class

public NumberDisplay(int rollOverLimit)

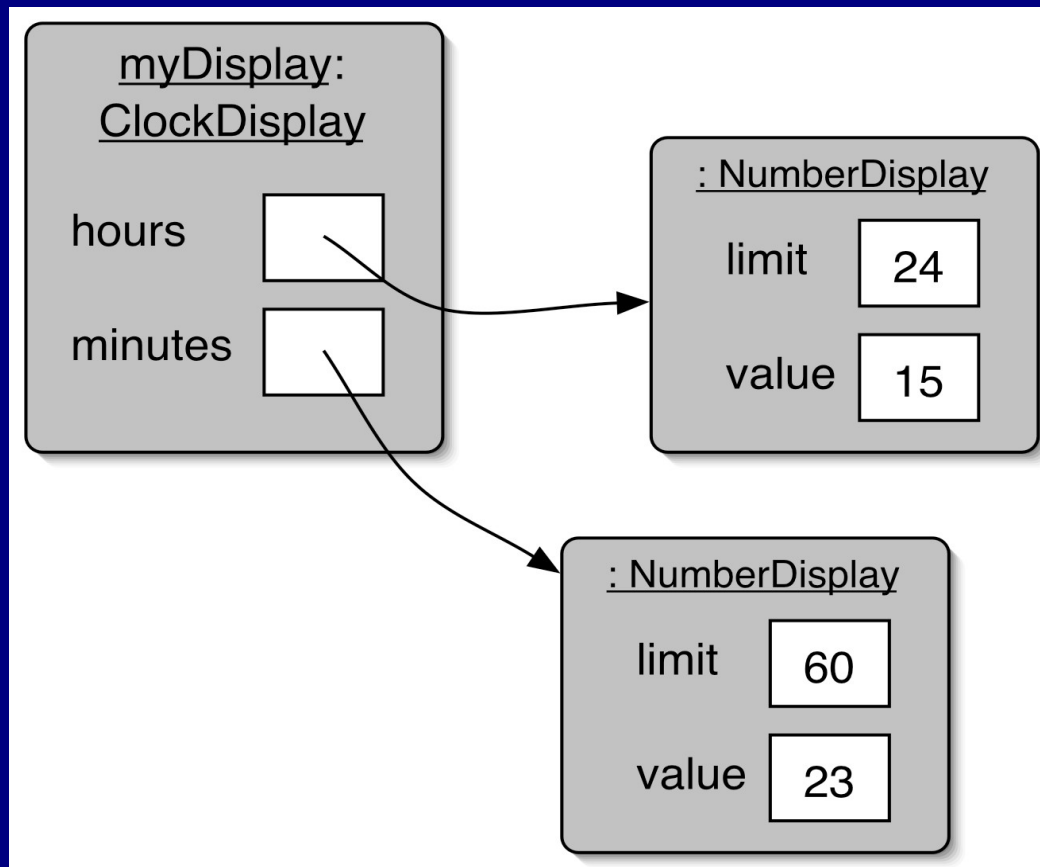
formal parameter

new NumberDisplay(24)

actual parameter



# ClockDisplay object diagram



# Method Overloading

- Multiple Constructors of ClockDisplay:
  - new Clockdisplay()
  - new Clockdisplay(hour, minute)
- It is common for class definitions to contain alternative versions of constructors or methods that provide various ways of achieving a particular task via their distinctive sets of parameters.
- This is known as **overloading**.



# Method calling

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





# Internal method

```
/**
 * Update the internal string that
 * represents the display.
 */
private void updateDisplay()
{
 displayString =
 hours.getDisplayValue() + ":" +
 minutes.getDisplayValue();
}
```



# Method calls

- internal method calls

```
updateDisplay();
```

```
private void updateDisplay()
```

- `methodName(parameter-list)`

- external method calls

```
minutes.increment();
```

- `object.methodName(parameter-list)`

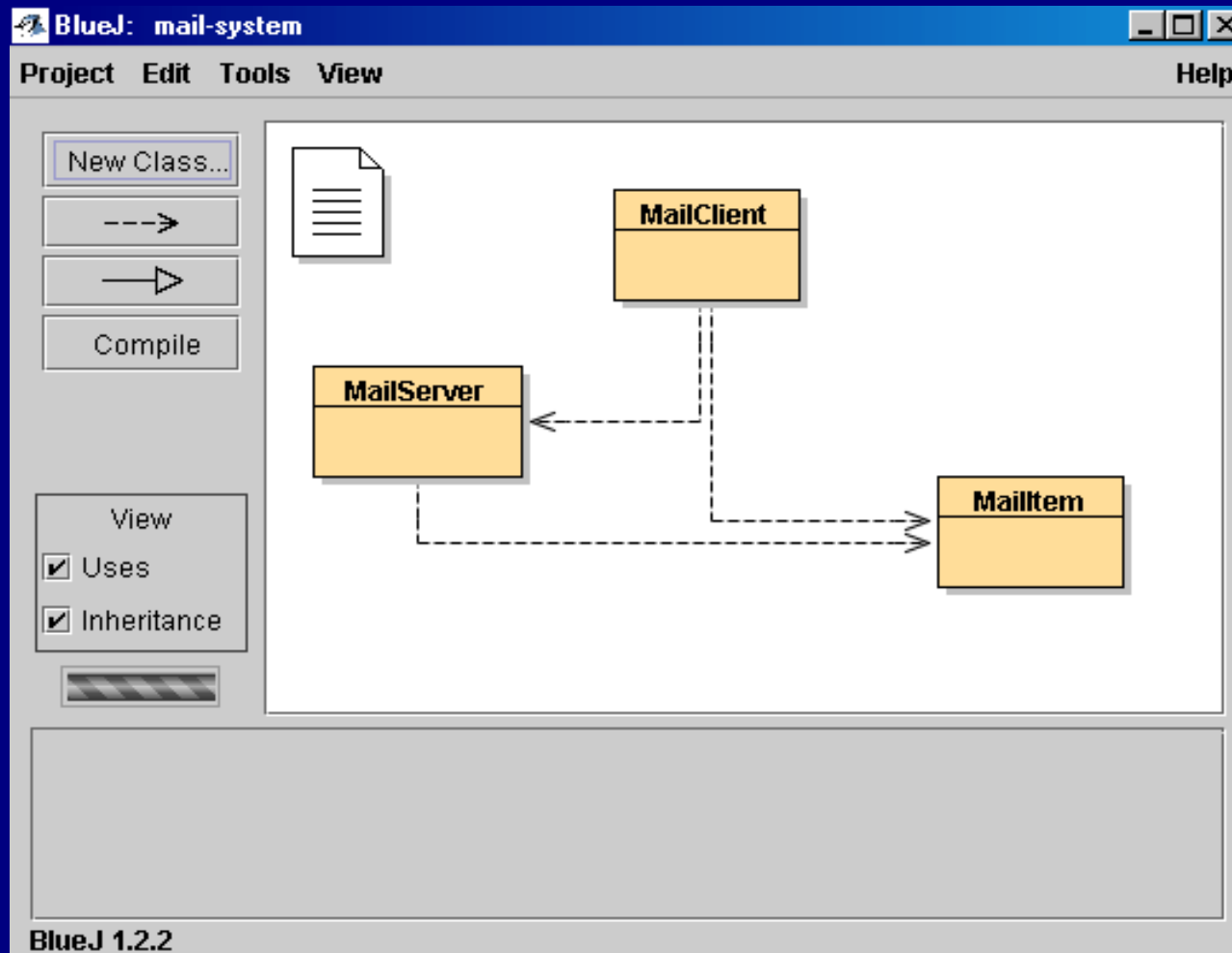


# Public and Private Methods

- Public methods:
  - `public void increment()`
  - can be called externally
- Private methods
  - `private void updateDisplay()`
  - can only be called internally
  - used for auxiliary methods



# The Mail System



# The this Keyword

```
public class MailItem
{
 private String from;
 private String to;
 private String message;

 public MailItem(String from, String to,
 String message)
 {
 this.from = from;
 this.to = to;
 this.message = message;
 }
}
```



# The this Keyword

- `this.from = from`
  - **name overloading**: the same name is used for two different entities: instance variable and formal parameter.
  - `this` is used to go out of the scope of the constructor to class level
  - `this` always refers to the current object.
  - can also used for methods
  - for internal methods calls and access to instance fields  
Java automatically inserts `this`
  - `updateDisplay -> this.updateDisplay`



# Concepts

- abstraction
- modularisation
- call-by-value
- call-by-reference
- logical operators/modulo
- this
- class/object diagram
- primitive types
- object types
- object creation
- overloading
- internal/external method calls
- private methods
- debugging

