Principles of Programming CM10227

Lecture D.7.: Java: Objects First

Dr. Marina De Vos
University of Bath
Ext: 5053

Academic Year 2012-2013

Resources

- Objects First with Java. David J. Barnes and Michael Kölling. Third edition
- How to Think Like a Computer Scientist: Java. `http://www.greenteapress.com/thinkapjava/`
- Big Java. Gay Horstman.
- Thinking in Java. Bruce Eckel's `www.mindview.net/Books/TIJ4`
- Sun Java Tutorials Series `http://java.sun.com/docs/books/tutorial/index.html`

Outline

1. Objections and Classes
2. Control Flow in Java
3. Object Interaction
4. Running Java Programs

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)

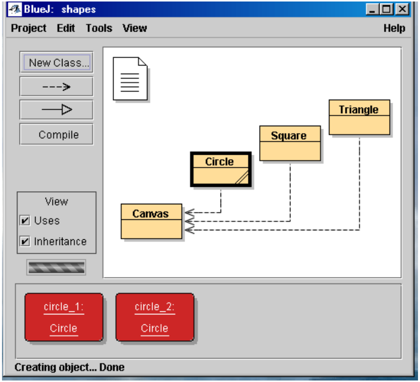
Objects represent individual instantiations of the class. Object are instantiated.

Exercise

TelephoneNumber
BankAccount
harry-potter-and-the-Philosopher-Stone
01225-38-5053
Book
leonWatts

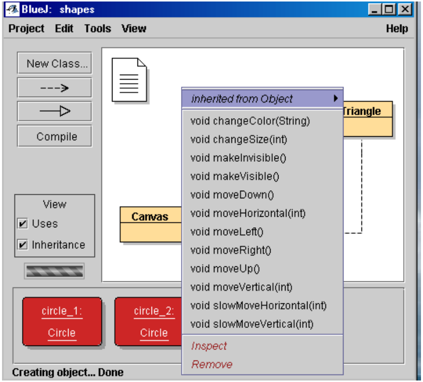
lord-of-the-rings
Diary
myDiary
marinaDeVos
Lecturer
myAccount

Objects and Classes



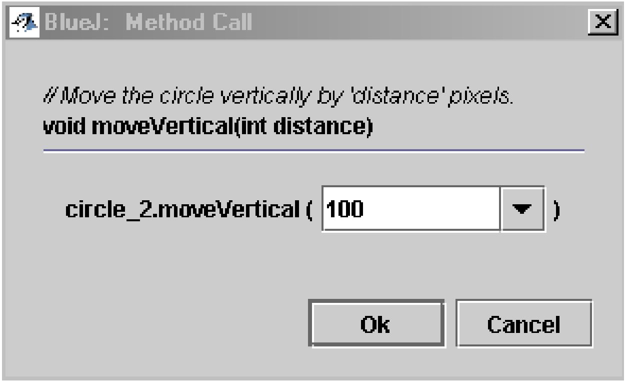
Creating object... Done

Things we can do with objects I



Creating object... Done

Things we can do with objects II



BlueJ: Method Call

// Move the circle vertically by 'distance' pixels.
void moveVertical(int distance)

circle_2.moveVertical (100)

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 method
- 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

Exercise: BankAccount

What are the methods should have BankAccount have?

Abstract Data Types, Objects and Classes

- In the Python lectures we discussed ADTs. They were implemented using nested functions. The outer function returned a lambda function allowing you to access the inner functions.
- A class is like this nested function
- An object is the result from calling the function, i.e. the lambda.
- Each time you call the outer function you will get a new lambda function and new internal data
- The methods correspond to the inner functions.

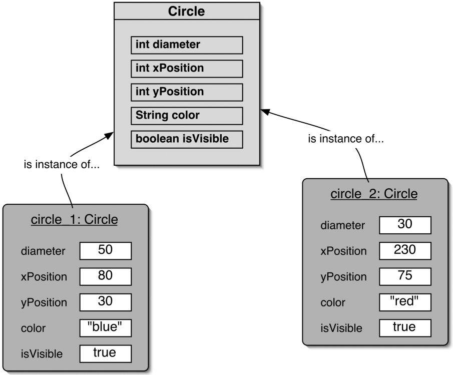
Data Types

- Parameters have types. A type defines what kinds of values a parameter can take.
- In Java you have to specify the type. This was not the case for Python.
- Defining a class defines a type
- In Java, everything has a type.
- Java is staticly typed language
- Examples of types: int, String, Circle, . . .

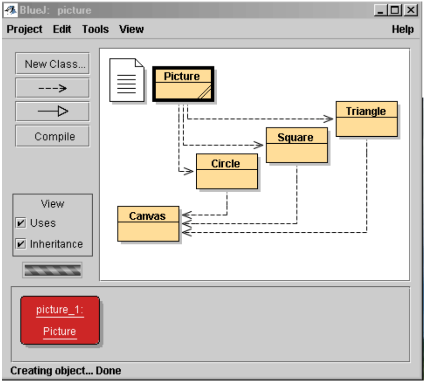
Other Observations

- many instances can be created from a single class
- an object has attributes: values stored in fields. (The data you encapsulate)
- 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.

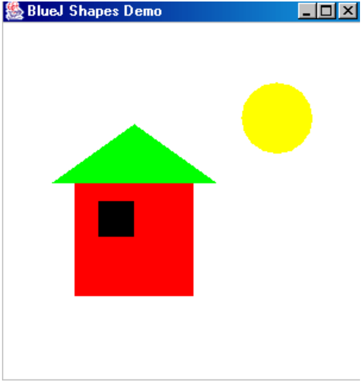
State


Circle
int diameter
int xPosition
int yPosition
String color
boolean isVisible
is instance of...
is instance of...
circle_1: Circle
diameter
50
xPosition
80
yPosition
30
color
"blue"
isVisible
true
circle_2: Circle
diameter
30
xPosition
230
yPosition
75
color
"red"
isVisible
true

Object Interaction



Creating object... Done



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
- 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

Glossary

Object	Instance	State
Method	Invocation	Class
Source code	types	fields
Attribute	parameter	return value

Main concepts to be covered

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

Ticket Machines An External/User 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: The State

```
private int price = 500;
private int balance = 0;
private int total = 0;
```

Resulting Methods: The Interface

```
public int getBalance()
public int getPrice()
public void insertMoney()
public void printTicket()
```

Ticket Machines An Internal/Programmer 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

```
/**
 * 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)
    {
        price = ticketCost;
```

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

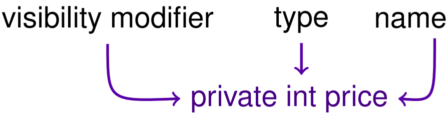
Comments/Documentation

- 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.
- Fields define the state of an object.
- Fields have an associated type.

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

    Constructor and methods omitted.
}
```



visibility modifier

type

name

private int price

Constructors

- Constructors create and 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.
- They Passing data via parameters

```
public TicketMachine(int ticketCost)

{
    price = ticketCost;
    balance = 0;
    total = 0;
}
```

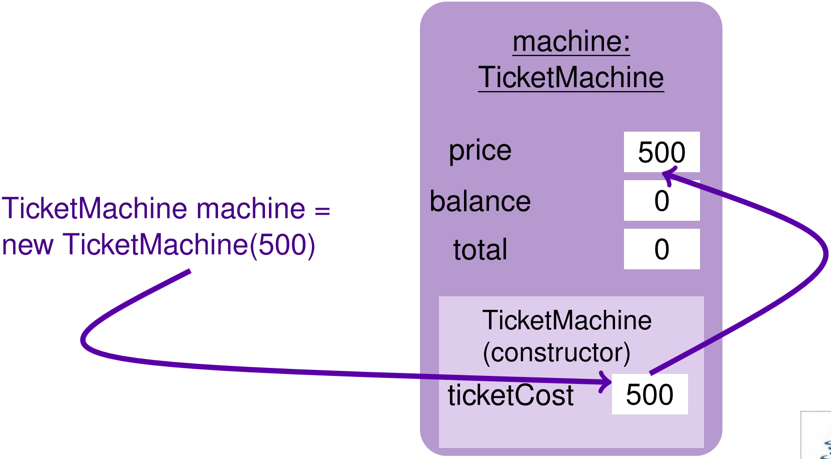
Creating Objects

- Constructors are used to create and initialise a new object

```
TicketMachine machine = new TicketMachine
    (500);
```

- This creates a new TicketMachine object and stores it a variable named machine which is of type TicketMachine.

Object Diagram



Parameters

- Just like in Python
- 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 constructor `TicketMachine(int ticketCost)` ticketCost is a formal parameter. When the constructor is called, `TicketMachine(500)`, 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.
- Concept the same as in Python.

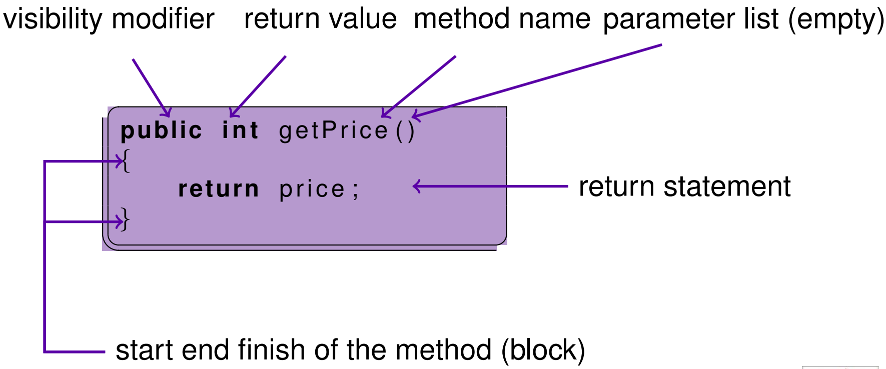
Assignment

- Similar to Python
- 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, TicketMachine, ...
- A variable stores a single value, so any previous value is lost.

Accessor Methods I

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

Accessor Methods II



visibility modifier

return value

method name

parameter list (empty)

```
public int getPrice()
{
    return price;
}
```

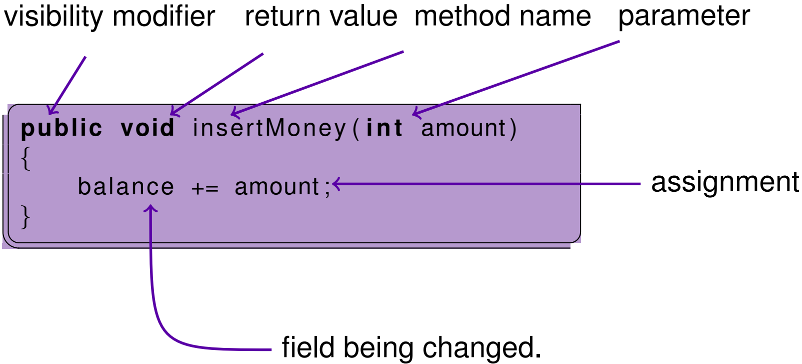
return statement

start end finish of the method (block)

Mutator Methods

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

Mutator methods



visibility modifier return value method name parameter

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

assignment

field being changed.

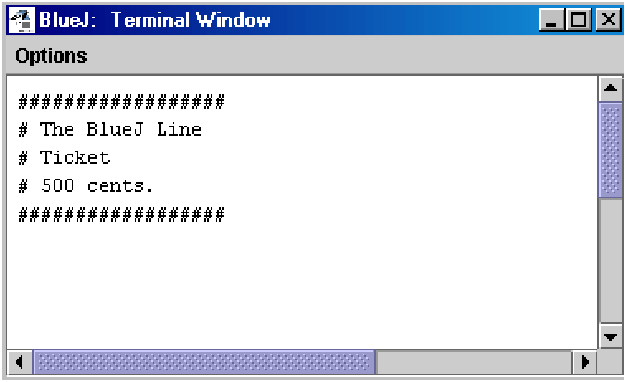
Abstract Data Types

- 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



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

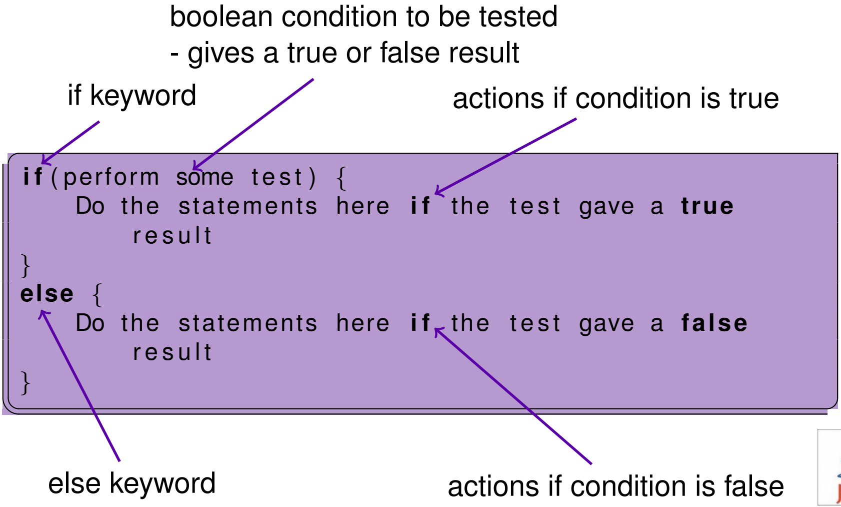
Reflecting on the ticket machines

- Their behaviour 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 behaviour.

Making choices

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

Making choices



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

if keyword

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

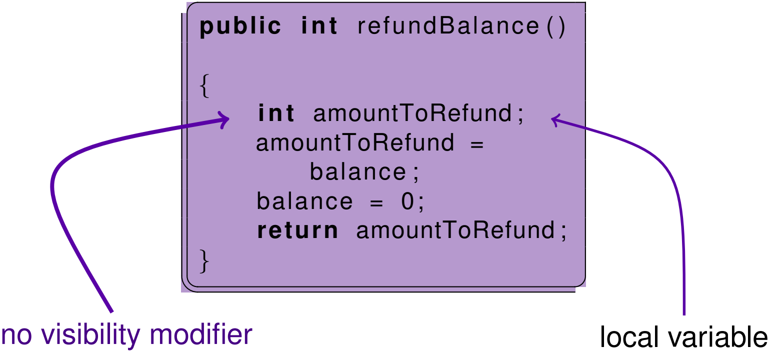
Boolean Tests

- $==$: 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.
  - A bit like global variables in Python
- 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.
  - Like function variables in Python

Local variables



```
public int refundBalance()

{
     int amountToRefund;
     amountToRefund =
          balance;
     balance = 0;
     return amountToRefund;
}
```

no visibility modifier

local variable

Review

- Class bodies contain fields, constructors and methods.
- Fields store values that determine an objects state.
- Constructors initialize objects.
- Methods implement the behaviour 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.
- 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.

Glossary

Terms	Instance variables	Local variables
Parameters	Formal Parameters	Actual Parameters
Scope	Lifetime	Assignment
Constructors	Methods	
If-statement	Object diagram	

Main concepts to be covered

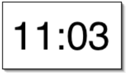
- Abstraction
- Modularization
- Class and Object Diagrams
- Call-by-reference and Call-by-value
- Overloading
- Internal and External method calls
- this keyword
- Debugging

A digital clock

Abstraction and modularization

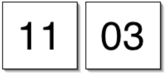
- Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.
- Modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

Modularizing the clock display



One four-digit display?

Or two two-digit displays



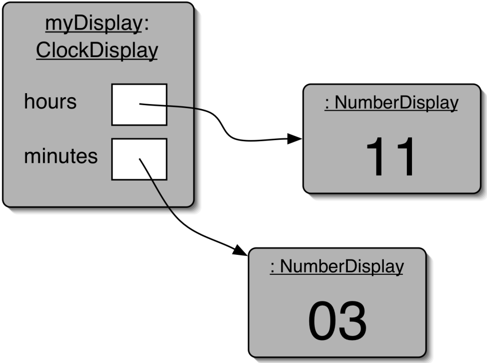
Implementation: NumberDisplay

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

    Constructor and
    methods omitted.
}
```

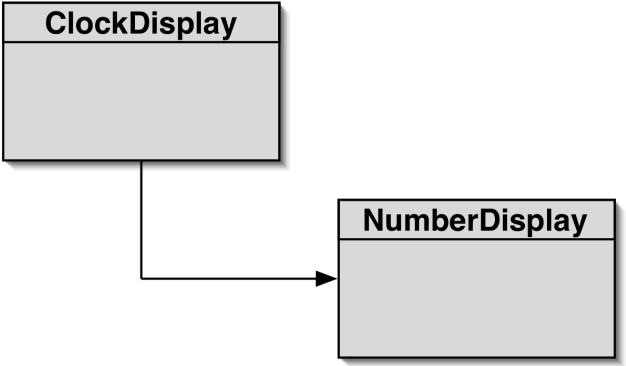
Implementation: ClockDisplay

```
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

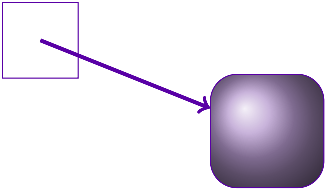
Primitive types vs. object types

- 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.
- This is the reason why Java is not a completely object oriented languages

Primitive types vs. object types

SomeObject obj;

Object Type

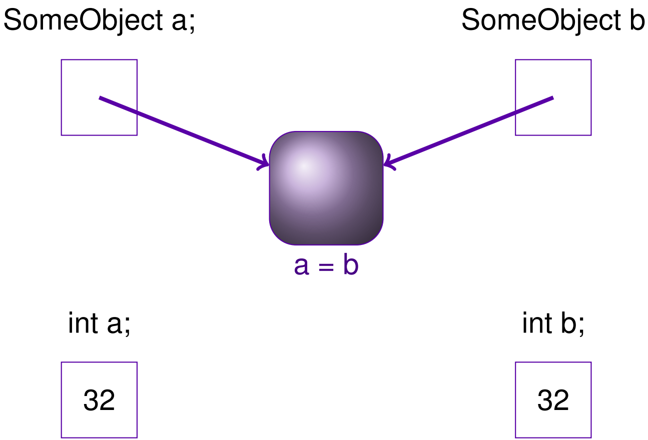


int i;

Primitive Type;

32

Primitive types vs. object types



SomeObject a;

SomeObject b

a = b

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: the caller gives the called method the ability to directly access to the callers data and to modify that data if the called method so chooses.
- Just like Python Java uses call-by-value
- For objects, the value is a reference to memory (like in Python)

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” + “ and Python” $->$ “Java and Python”
  - “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

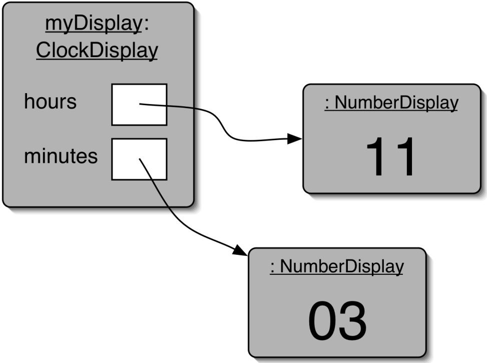
Objects creating objects

```
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

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 constuctors 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

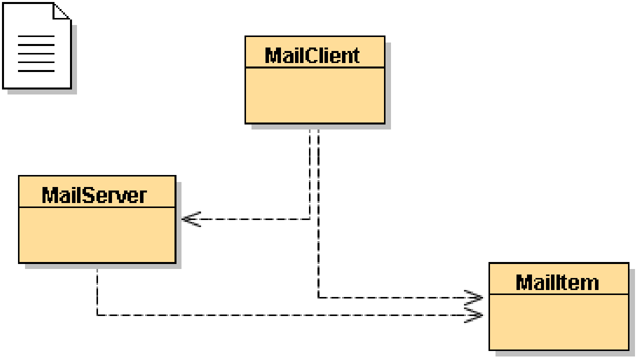
```
minutes.increment();
```

  - 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;
    }
```

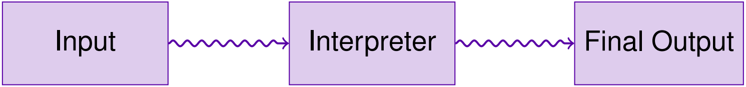
The this Keyword

- **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

Glossary

Abstraction	Modularisation	this
Call-by-value	Call-by-reference	Class diagram
Logical Operators	Modulo	Object diagram

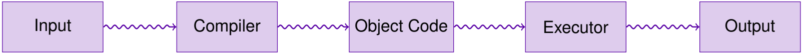
Interpreted Languages



The interpreter reads in the source code ...

... and the result appears on the screen

Compiled Languages



Input → Compiler → Object Code → Executor → Output

The compiler reads in the source code ...

... and generate the object code ...

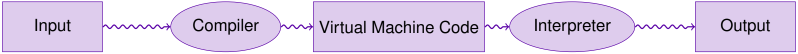
... you execute the code (one way or the other) ...

... and the result appears on the screen

The Programming Language Java

- The Java language is both compiled and interpreted.
- Instead of translating Java programs into a machine language, the Java compiler generates Java byte code for its Virtual Machine
    - Byte code is easy (and fast) to interpret, like machine language,
    - but it is also portable, like a high-level language.
- Thus, it is possible to compile a Java program on one machine, transfer the byte code to another machine over a network, and then interpret the byte code on the other machine.
- This ability is one of the advantages of Java over many other high-level languages.

Java: Compile - Interpret



The compiler reads in the source code ...

... and generate the byte code for the virtual machine ...

... the code is interpreted ...

... and the result appears on the screen

Compiling and Running Simple Program I

- A simple classical example is the Hello World program.

```
public class HelloPrinter
{
   public static void main(String[] args)
   {
      // Display a greeting in the console window

      System.out.println(”Hello␣World”);
   }
}
```

- The filename should match the name of the class with the extension .java. In this case, HelloPrinter.java
- Java is case sensitive, just like Python.

Compiling and Running Simple Program II

- To run the code:
  - we need to compile it: javac HelloPrinter.java
  - This will generate a file HelloPrinter.**class**, containing the virtual machine byte code
  - We can now run the code: java HelloPrinter

```
Hello, World
```

- The contruct **public static void** main(String[] args) defines the method called main
- Every Java application must have a main method.
- The parameter String[] args is required. args will contain the command-line arguments.
- The keyword **static** means it is a class method rather than an object method. main has to be static.

Compiling and Running Programs Consisting of Multiple Classes

- Compile all classes, using javac. On the linux system you can use javac ∗.java to compile all .java files in one go.
- To run the program, you need to use java on the class that contains the main method.

Implementing a Test Program I

- The purpose on a test program is to verify that one or more methods have been implemented correctly
- A test program calls methods and checks that they return the expected results.
- It contains the following steps:
  1. Provide a tester class
  2. Supply a main method
  3. Inside the main method, create one or more objects
  4. Apply methods to the objects
  5. Display the results of the method calls - if needed
  6. Display the valued that you expect to get - if possible

Implementing a Test Program II

- Consider the Shapes project. It contains allows you to draw circles, squares and triangles on a canvas.
- To this extend it contains the classes: Circle, Squares, Triangle and Canvas
- To test if the implementation is correct we can write a test class

```
public class ShapesTest
{
   public static void main(String[] args)
   {
        Canvas c = Canvas.getCanvas();
        Circle c1 = new Circle();
        Square s1 = new Square();
        Triangle t1 = new Triangle();
        c1.makeVisible();
        s1.makeVisible();
        t1.makeVisible();
        ...
   }
}
```

Implementing Applications

- the main method of your application class should be relatively short
- normally a few objects are created and a few methods are invoked.
- the invoked methods will determine the behaviour of your application.

Glossary

Compiler	Virtual Machine	Byte Code
java	javac	main method
test program		