

# CITS1001

## SOFTWARE ENGINEERING WITH JAVA

---

Semester 2, 2022: Introduction and Admin

Dr Max Ward

Computer Science & Software Engineering

# Teaching Team

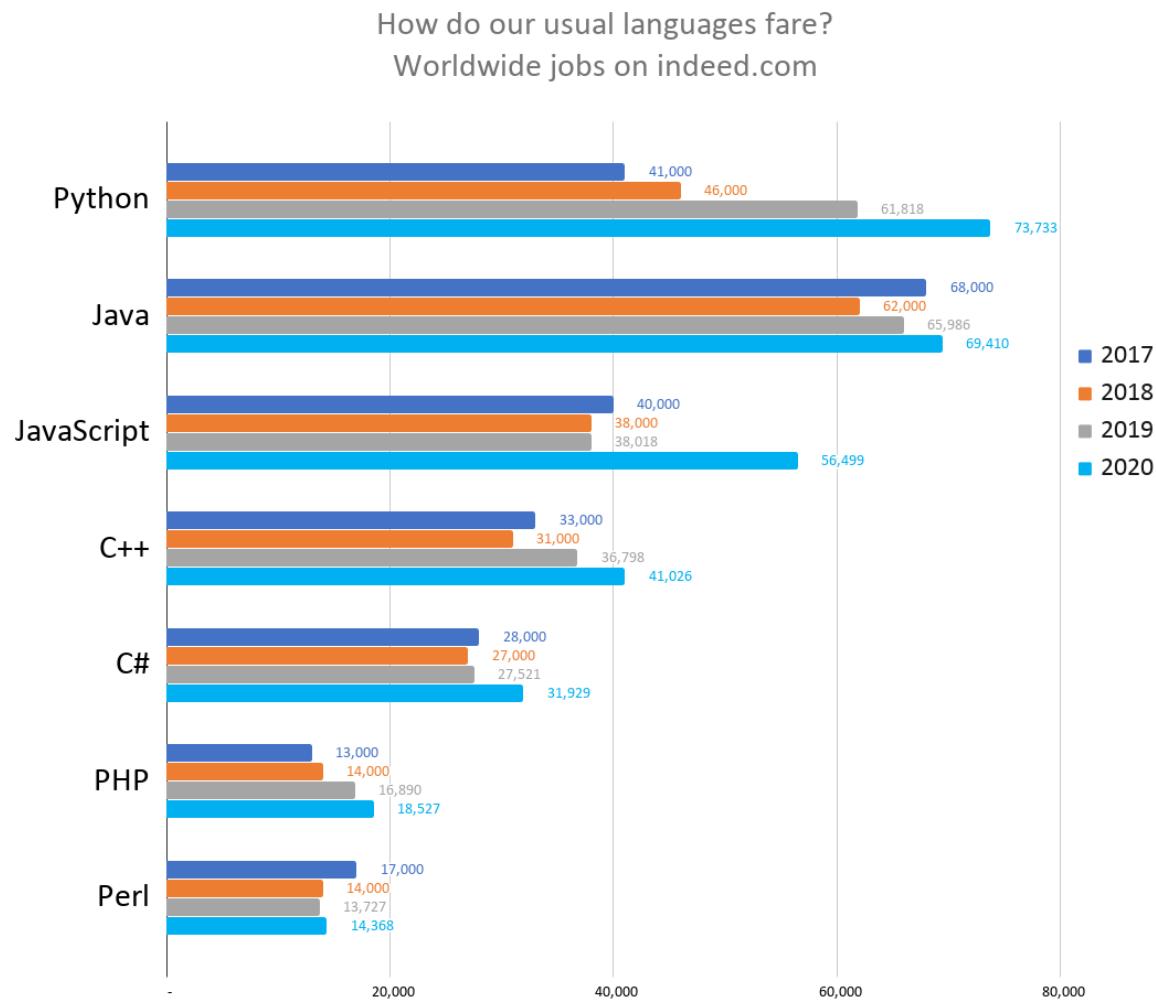
- Unit Coordinator & Lecturer
  - Dr Max Ward
- Guest Lecturers
  - Andrew Gozzard (week 2)
  - Prof Amitava Datta (week 3)
- Lab facilitators
  - Caitlin Woods
  - Marcell Szikszai
  - Maira Alvi
  - David Charkey
  - Harry Mueller
  - James Arcus

# Unit objectives

- In CITS1001 you will learn how to read, create, and understand simple computer programs in the Java programming language
- Today's lecture will give you an overview of the content and organisation of the unit, and what you will have learned by the end of the semester

# Java in the Job Market

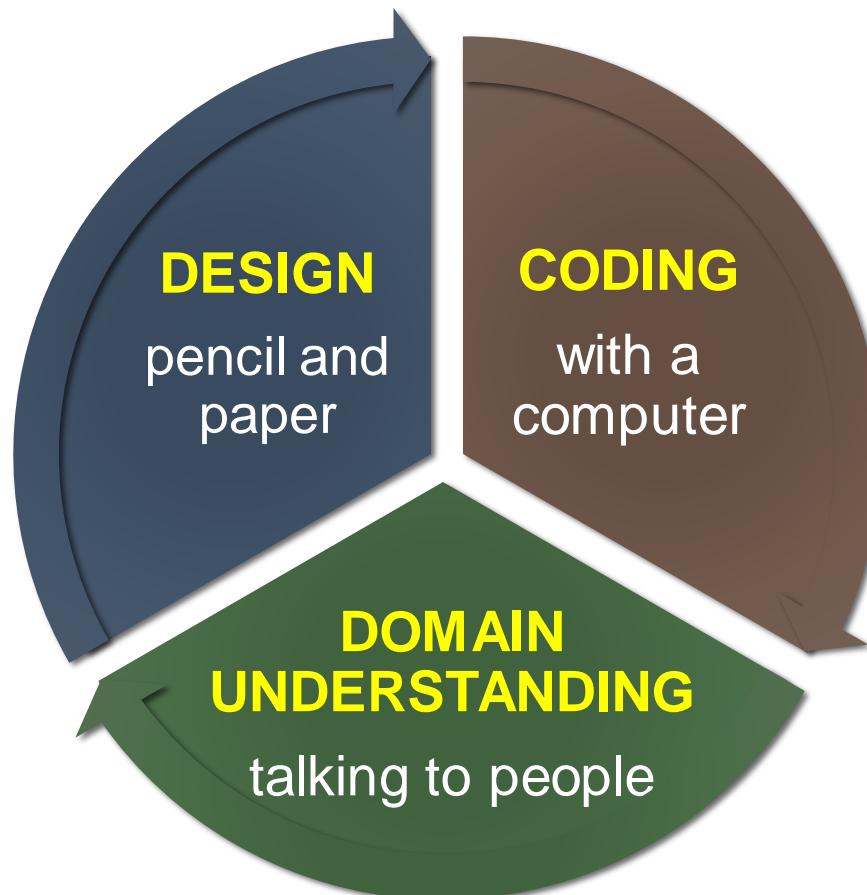
- Demand for Java has been more stable over the years
- Python is mainly for scripting, quick prototyping or used as a front end with C libraries
- In general, demand for programmers is rising



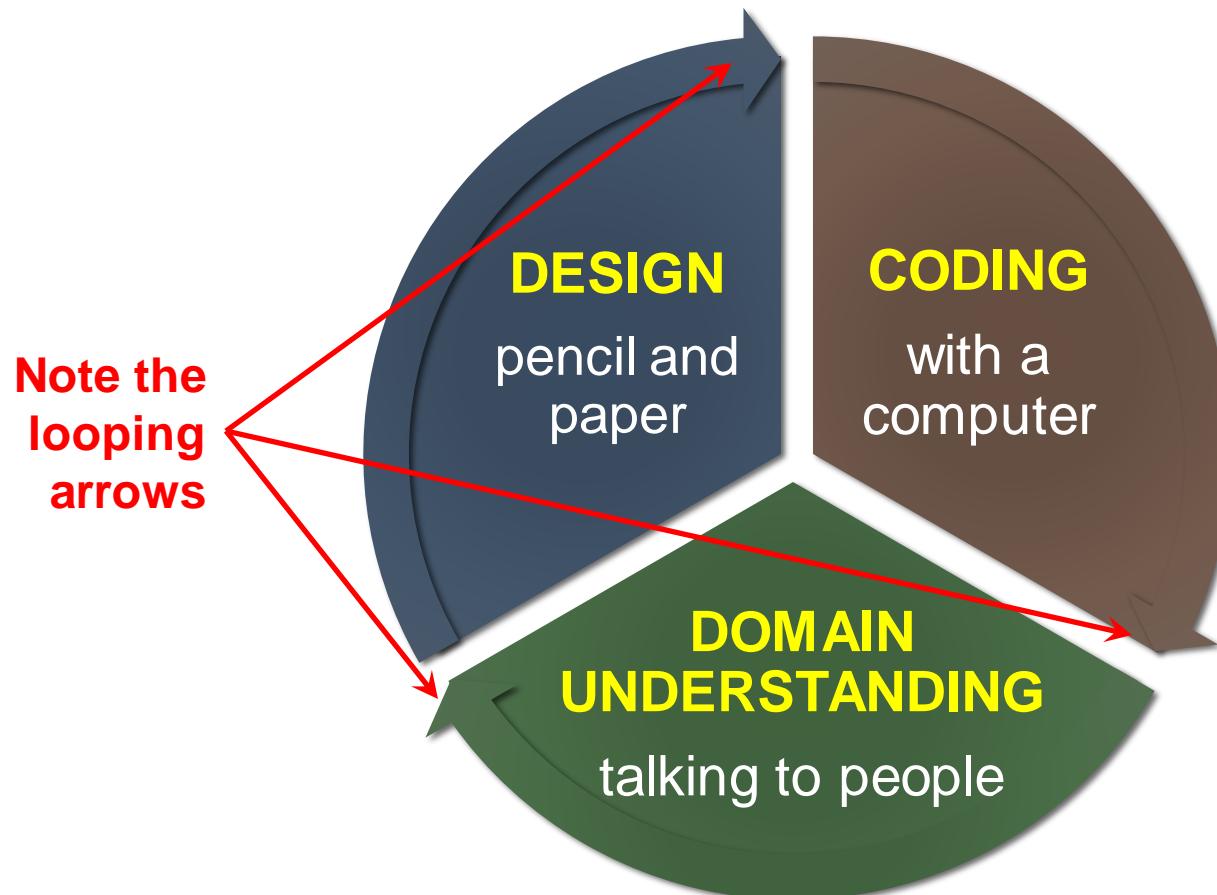
# Some terminology

- **Programming** is the process of building an executable model of part of the world inside the computer
  - Before you can build such a model, you have to understand both the relevant part of the world, and how it can be represented
- **Java** is one of the most widely-used programming languages currently found around the world
  - Smart phones, web software, financial software, embedded code, etc.
- **Software engineering** is the application of engineering principles to the development of software
  - Systematic, disciplined, quantified, validated, verified, secure, robust

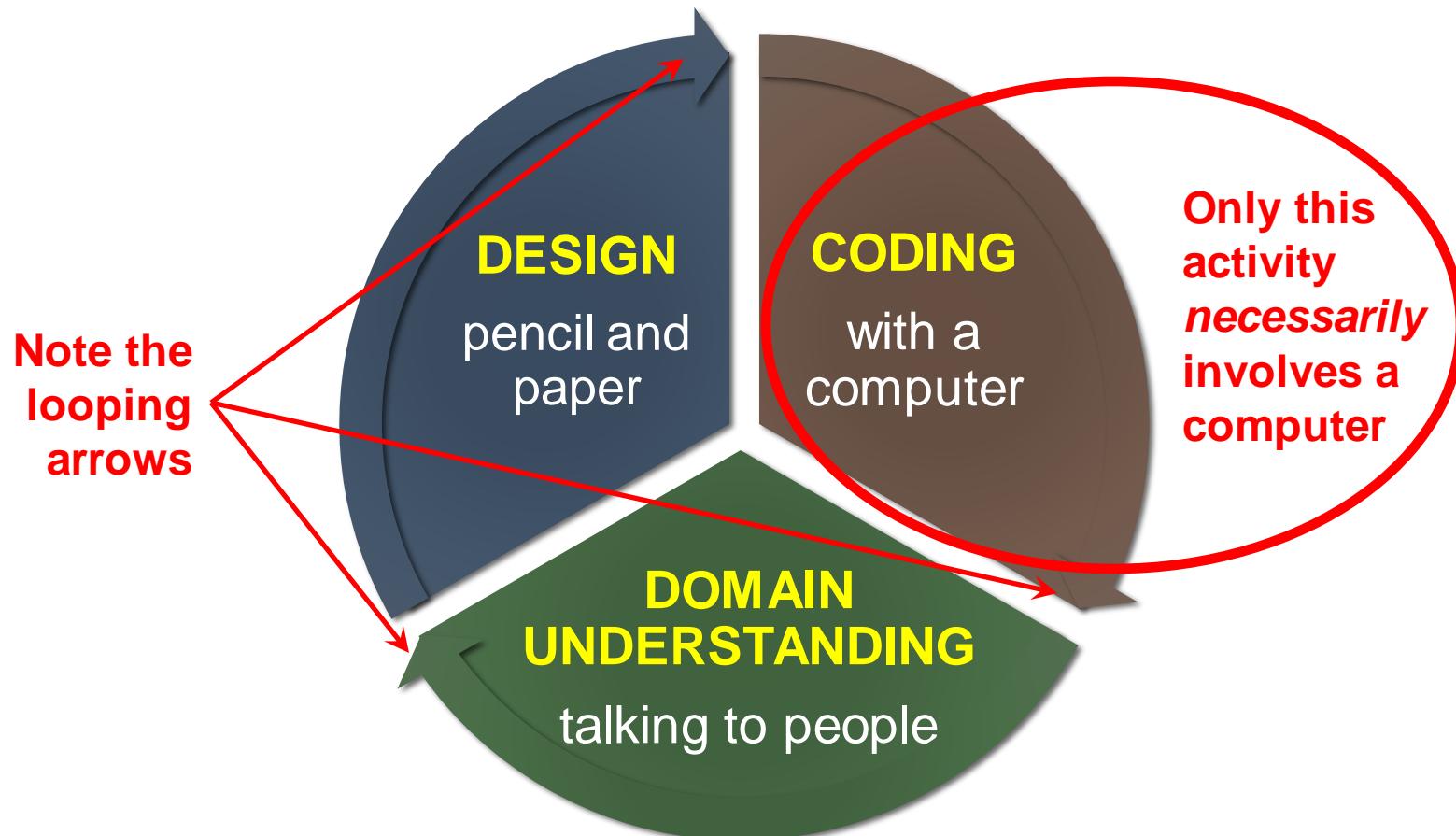
# What do software engineers do?



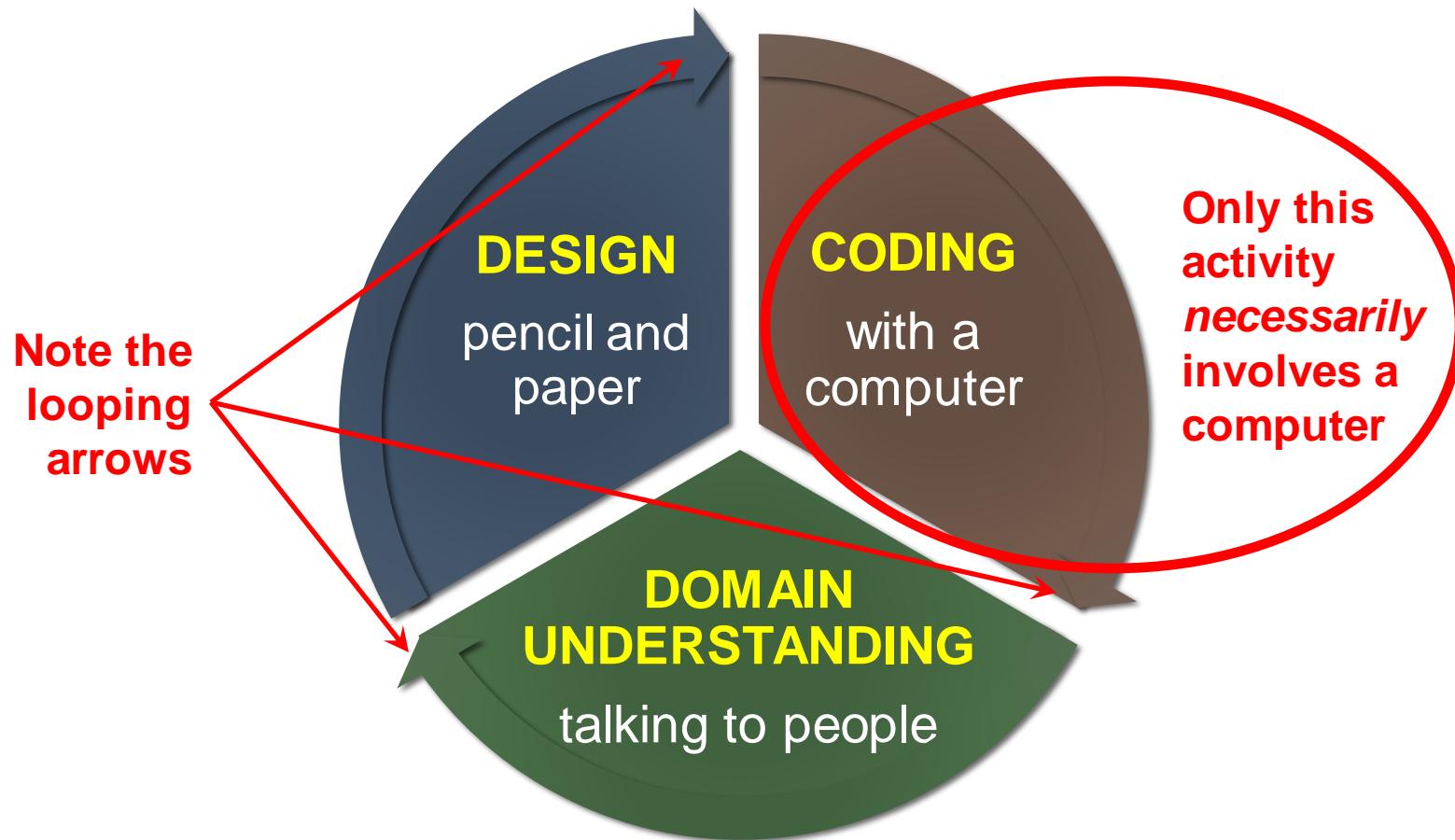
# What do software engineers do?



# What do software engineers do?



# What do software engineers do?



*Computer Science is no more about computers than Astronomy is about telescopes. (Edsger Dijkstra)*

# Unit philosophy

- CITS1001 is a first programming unit
  - No prior programming experience is assumed
  - But students' backgrounds vary widely!
- No one can turn you into a productive programmer in a single three-month course
- CITS1001 is the first in a sequence of units giving you
  - A detailed understanding of programming and software engineering
  - Excellent problem-solving skills
  - Exposure to a wide range of topics in ICT
- If your plan is to do only one UWA programming unit, it may be wise to consider instead the unit  
*CITS1401 Computational Thinking with Python*

# Delivery Mode

- CITS1001 will be delivered face 2 face
- As well as online for those registered online
- I encourage you to enroll for face 2 face if possible

# Lectures

- Lectures will be recorded and available from the LMS
- Tuesday and Thursday
- The lecture slides will also be available from the LMS, at the start of each week
- Lectures will introduce concepts in software engineering and in programming using Java, and will illustrate those concepts via worked examples
- Lectures may occasionally include revisions, description of the projects or case studies
- This will be clearly indicated on the LMS page and lecture slides

# Workshops

- 10AM-11AM on Fridays starting in Week 2
  - Same lecture theatre
- The workshops will not introduce new material but include
  - Student-driven questions
  - Additional examples complementing the lectures
  - Support for assessments
- Aimed principally at students who feel they need extra support
  - All questions are welcome, but priority will be given to more-basic questions
- Workshops will also be recorded and available from the LMS

*The only stupid question is the one that isn't asked!*

# Labs

- The labs are where you will practice creating, running, and testing programs in Java
- Each week's exercises will be on a labsheet on the LMS
- Labs will start in Week 2
- Face 2 face labs will be in CSSE Lab 201
- Several sessions are available, each two hours long
  - A lab facilitator will be available to answer questions and give advice
- Enroll in one session but feel free to attend as many sessions as you like
  - Or to do the work at other times

*Learning to program is like learning to swim: you have to DO IT!*

# Labs (Online)

- Online labs are via MS Teams
- Available only to those students who have registered for online course
- A lab facilitator will be online to answer your questions
- You will use the same LMS labsheets as the face 2 face students use

# help1001

- An online discussion forum, based on the philosophy of

*READ FIRST: if the answer is not there, THEN POST.*

- <https://secure.csse.uwa.edu.au/run/help1001>
- Asking questions is useful
  - Normally the quickest way to get help
  - Sometimes just formulating a question properly is enough for you to realise what the answer is
  - ***Use meaningful subject lines please, to facilitate searching***
- Answering questions is useful
  - Explaining something helps you to understand it better
  - And it generates good karma too!
- Join lab and workshop sessions also to get help

# Announcements

- Announcements will be made in two places
  - On the LMS
  - On *help1001*

*When an announcement has been made in these two places, we will assume that you are aware of it.*

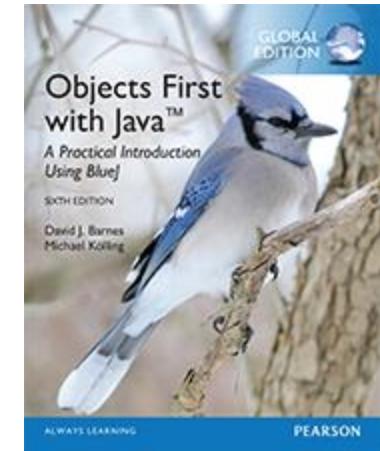
# Assessment

- Assessment is based both on understanding concepts and on creating systems
- Mid-semester Test (15%)
- Two programming projects (10% and 15%)
  - Weeks 6–7 and Weeks 10–12
- Final exam (60%)

*Everyone here is capable of passing CITS1001, but history suggests that 15–20% of the group will fail: which subset do you want to be in?*

# Text and other resources

- The text in CITS1001 is *Objects First with Java*, Barnes and Kölling, sixth edition
  - Earlier editions should be fine, but note that chapter numbers may vary
- There are also free texts available online
  - e.g. at <https://greenteapress.com/wp/>
- In fact there are thousands of Java resources out there
  - Go and look!
  - No explanation from me will be optimal for all of you
  - If you find something that is useful to you, please post a message on *help1001* so that others can benefit too

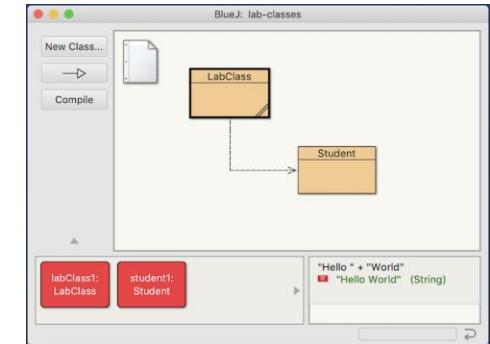


# BlueJ

- Programming activities in CITS1001 will be based on the BlueJ IDE
  - “A free Java development environment designed for beginners”



- You can download BlueJ for your home machine for free at [www.bluej.org](http://www.bluej.org)
  - It is available for all major platforms
- Of course you can use any Java system you like at home
  - But all work submitted for assessment will be tested on BlueJ



```
    /**
     * Add a student to this LabClass.
     */
    public void enrollStudent(Student newStudent)
    {
        if(students.size() == capacity) {
            System.out.println("The class is full.");
        }
        else {
            students.add(newStudent);
        }
    }
}
```

# LMS (Learning Management System)

- Let us see the LMS page
- LMS is your one stop for all learning material
  - Lectures
  - Lab sheets
  - Video recordings
    - Lecture Capture System (current recordings)
    - Pre-recorded videos

# Seeking help

- There are many avenues for getting help in CITS1001
  - Ask questions in workshop sessions
  - Contribute to the workshops
  - Attend lab sessions and ask facilitators
  - Post on *help1001*
  - Make use of consultation times
  - [www.student.uwa.edu.au/learning](http://www.student.uwa.edu.au/learning)
  - The world-wide web
- The only stupid question is the one that isn't asked!
  - In a group this size, if you don't understand something, certainly there will be others in the same situation



# UWA compulsory online modules

- All UWA students are required to complete three online learning modules
  - Academic Conduct Essentials
  - Communication and Research Skills
  - Indigenous Study Essentials
- [www.student.uwa.edu.au/learning/resources](http://www.student.uwa.edu.au/learning/resources) has the details of these modules

# Things to do this week

- Set up your UWA computer account
- Sign up for a lab class
  - But labs start on **Monday** in Week 2
- Review these lecture notes
- Get a copy of the text book
  - And start reading it!
- Familiarise yourself with the unit on the LMS
  - So you can find information quickly when you need it
- Install BlueJ on your home computer



# What will you learn this semester?

- I will show you what students who apply themselves are able to construct at the end of CITS1001
  - Apologies if you have seen this previously
- *1010!* is a kind of 2D Tetris
  - <https://poki.com/en/g/1010-deluxe>
  - <https://en.wikipedia.org/wiki/Tetris>
  - Apps are also available
- Project 2 in 2015 was to construct a framework for playing *1010!*
  - Bonus marks were also available for extending the project further
  - Have a look at what students were able to achieve after only one semester
  - The BlueJ *1010!* project is available on the LMS

# Remember to

- Attend all lectures or watch them online
  - In the same week
- Read the text book
- Understand the concepts
- Practice programming
- Complete all lab sheets yourself
- Do your project yourself

# CITS1001

## 2. OBJECTS AND CLASSES

---

*Objects First With Java, Chapter 1*

# Lecture essentials

- Complexity and abstraction
- High-level decomposition vs. Low-level details
- Objects vs. Classes
- Compile-time construction vs. Run-time execution
- The BlueJ view

# Managing complexity

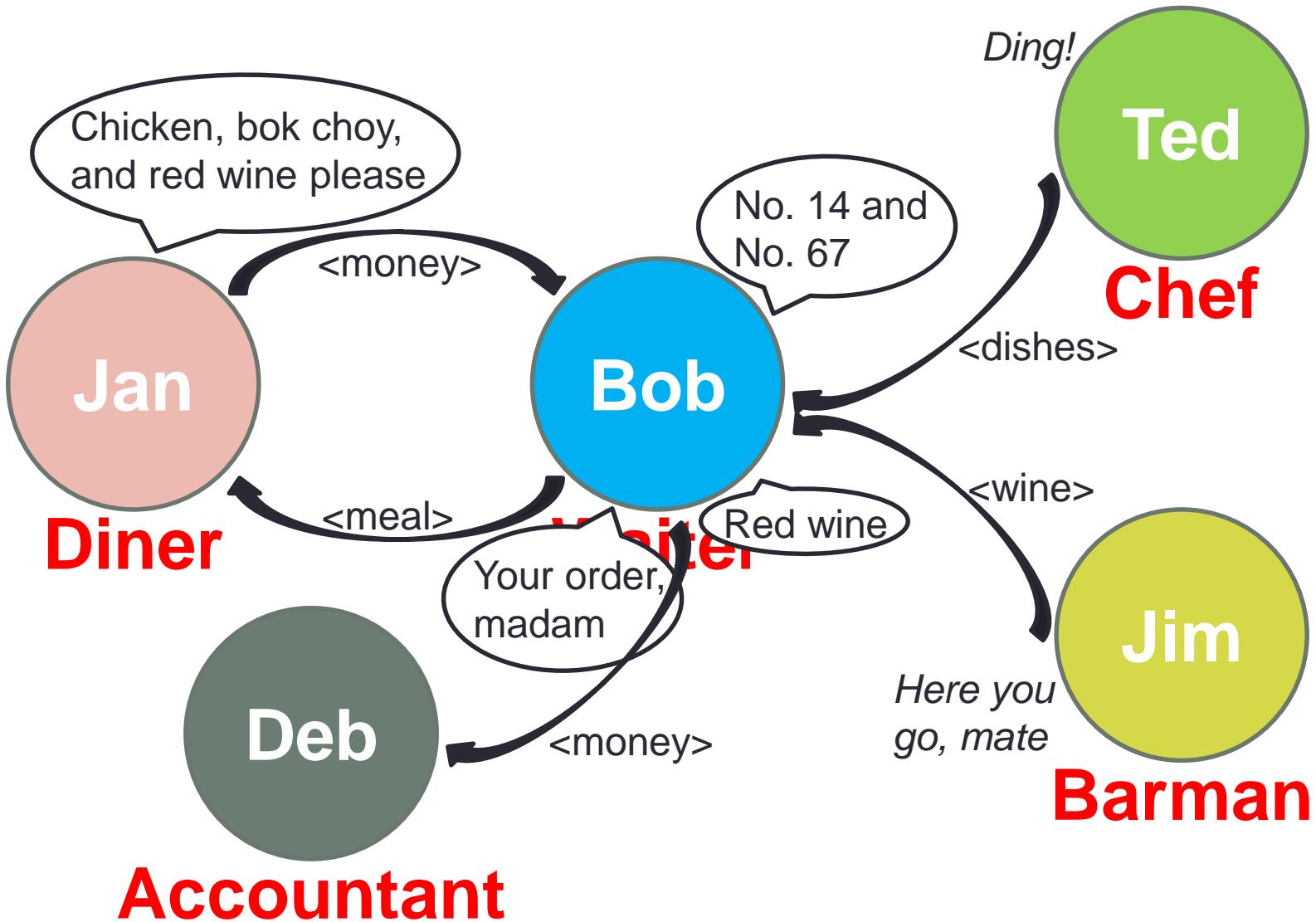
- Writing software is about managing *complexity*
  - Large programs are among the most complex artifacts ever created
- There are questions at (at least) two separate levels
  - How is the software divided up (*decomposed*) into components?
  - How does each component work?
  - These can be thought of as “high level issues” and “low level issues”
- The low-level question is about how code and data are structured; we will come to this soon
- In an object-oriented language like Java, the high-level decomposition is managed through the related concepts of *objects* and *classes*, which is where we start
- Consider first a real-world operation like a restaurant...

# A restaurant

- Operating a restaurant is complicated; there're people, food, money, real estate, regulations, ICT, furniture, facilities, etc.
- This complexity is managed partly by the specialization of people into different roles, e.g.
  - Diners order and consume meals
  - Chefs prepare dishes
  - Waiters take orders and bring food to the tables
  - Bar staff prepare and serve drinks
  - Accountants manage incomings and outgoings of money
  - Lawyers, IT professionals, trades people, etc. do other tasks

*Each type of person provides a narrow range of services. The restaurant involves the co-operative interaction of all the restaurant staff and clients.*

# Restaurant roles interacting



# Observations of the restaurant

- As a diner, I can interact with a waiter like Bob *simply by knowing that he is a waiter*
  - I know that he will take my order
  - I know that he will bring me food
  - I don't need to have met Bob before, or know anything else about him
- All waiters share common attributes and actions
  - An example attribute would be their name
  - An example action would be taking orders
- Similarly if we consider e.g. dogs, when I meet Rover
  - I know he will have fur, and legs, and teeth
  - I know he might bark, or bite, or wag his tail
  - I know all this before I meet Rover, simply by knowing his type

# Observations contd.

- Also, when I place an order with Bob, I know he will (eventually!) bring me the appropriate food
- Importantly, *I don't need to know how he does this*
  - Do they record orders on paper, or on a tablet device?
  - Do they serve food on crockery, or in a box?
  - Do they use metal cutlery, or plastic?
- It doesn't matter to me!
  - (Well mostly... of course some details might matter)
- This is the principle of *abstraction*
  - When we invoke an operation, we care *what job it does*, we usually don't care *how it does that job*
  - We care about the operation's *external* effects, but not about its *internal* workings

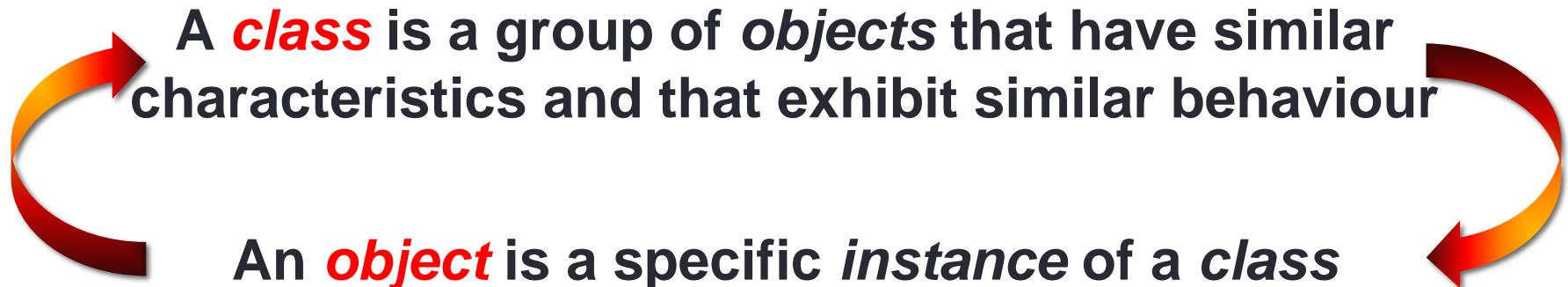
# Modelling the restaurant

- If we model this part of the world in Java
- Each role corresponds to a *class*
  - e.g. there would be a class Waiter, a class Diner, and a class Barman
- Each individual corresponds to an *object* belonging to a class
  - e.g. the class Waiter might have the *instances* Bob, Jo, and Lynn
  - e.g. the class Diner might have the instance me
- As I am a Diner, Bob *et al.* are all just waiters to me
  - e.g. when I want to order food from the kitchen, I don't do it myself; I ask a waiter-object to do it for me
  - e.g. when Bob needs to fill a drink order, he doesn't do it himself; he asks a barman-object to do it for him

# Modelling the restaurant contd.

- Objects in a given class perform a narrow range of well-defined services
  - The system operates through the co-operative interaction of many objects belonging to multiple classes
- When an object needs a service that is performed by another class, it calls upon a different object to perform that service
  - e.g. a diner-object calls upon a waiter-object to fetch some food
  - e.g. a waiter-object calls upon a chef-object to cook some food
- This is the “high-level” view of the restaurant
  - How the overall organization is decomposed into different roles
- The “low-level” view is how each object performs its job(s)
  - Much more on this later in the unit

# Objects and classes



A **class** is a group of **objects** that have similar characteristics and that exhibit similar behaviour

An **object** is a specific *instance* of a **class**

- Objects represent actual “things” from the real world, or from some problem domain
  - e.g. the red car in the car park
  - e.g. the lecturer talking to the group now
  - e.g. you!
- Classes represent all objects of a certain kind
  - e.g. Car, Lecturer, Student

# Operational relationship

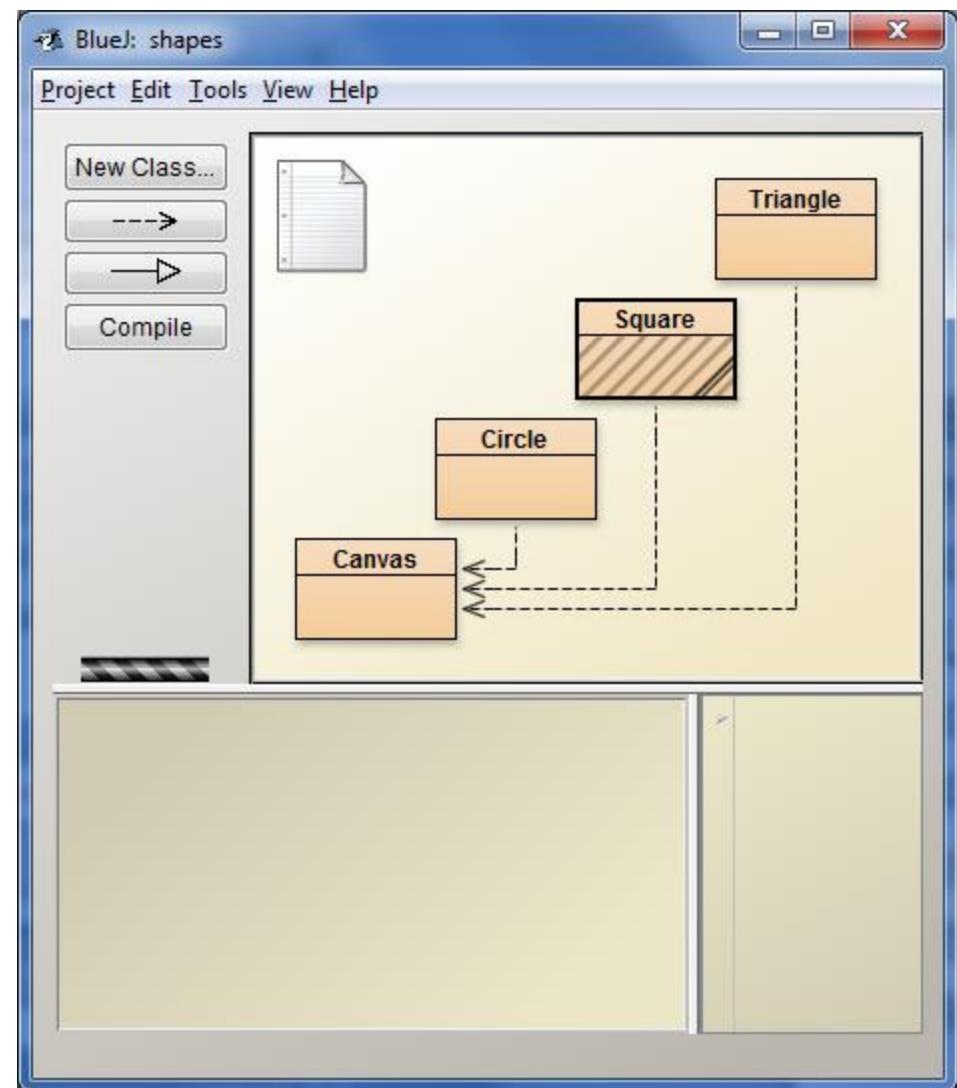
- The relationship between objects and classes is fundamental in an object-oriented language like Java
- It also reflects the distinction between
  - *compile-time*: when you are constructing the program
  - *run-time*: when you are executing the program
- In your source code at compile-time
  - You write a class to describe what a Waiter is and what a Waiter does
  - These attributes and actions apply to all Waiters created in the program
- When you execute the code at run-time
  - Your class is used to create Waiter-objects Bob, Jo, *et al.*
  - Possibly (usually!) many such objects
  - And those objects invoke operations to execute the code

# Operations contd.

- This point is illustrated by the lab exercises in Week 2
  - You will take some pre-defined classes and use them to create and manipulate objects
- The following slides show some stills from BlueJ operation to illustrate some of these points

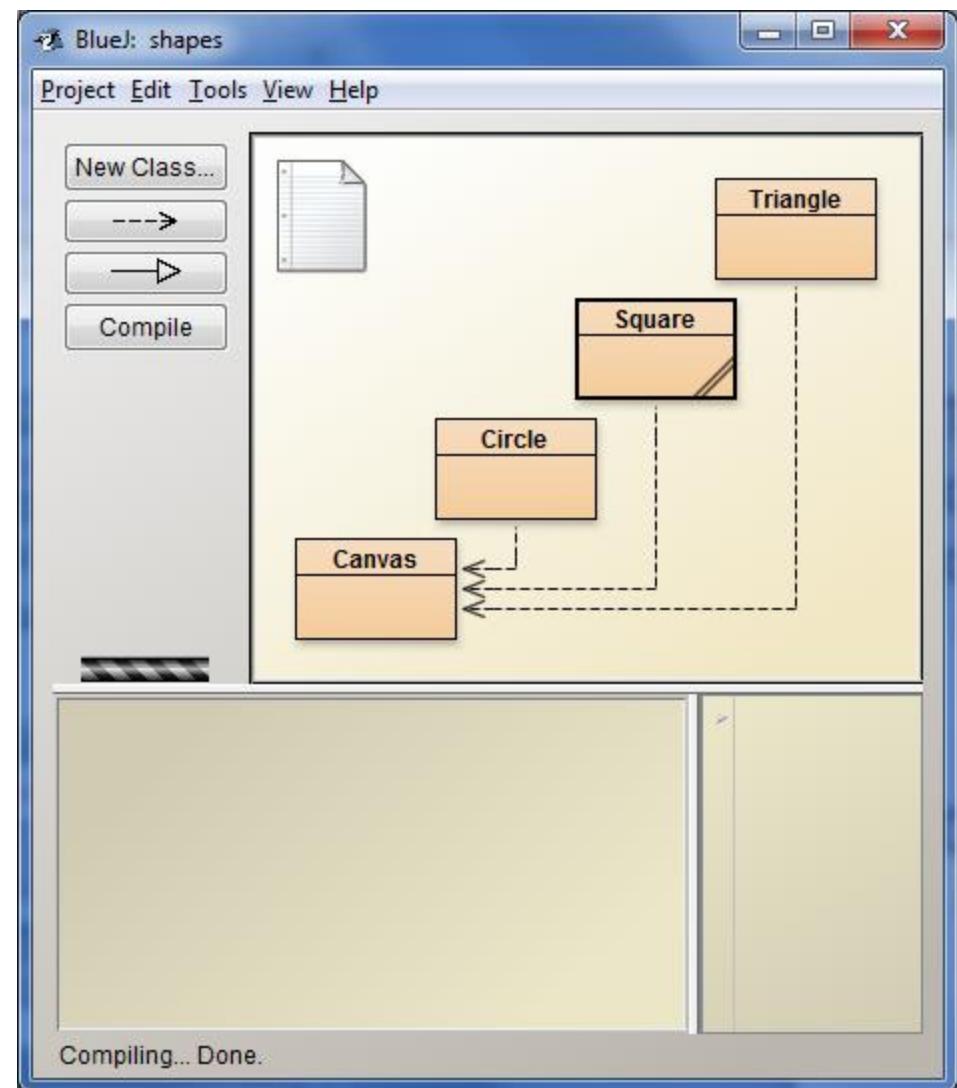
# The *shapes* project loaded into BlueJ

- Each rectangle represents one class in the project
- Lines represent connections (or *dependencies*) between classes
- If any box is shaded, that class needs to be compiled
  - Click the Compile button



# The source code for the Circle class

- To see the code for class Circle:  
double-click on Circle
  - Use the scroll bar(s) if you want to inspect all of the code
- Don't panic!
  - We will start exploring these types of details next week



# The source code for the Circle class

- To see the code for class C:  
double-click on C
  - Use the scroll bar(s) if you want to inspect all of the code
- Don't panic!
  - We will start exploring these types of details next week

The screenshot shows the BlueJ IDE interface with the title bar "BlueJ: shapes". A window titled "Circle - shapes" is open, displaying the source code for the Circle class. The code is color-coded: comments are in blue, strings are in red, and other identifiers are in black. The code defines a public class Circle with private attributes for diameter, xPosition, yPosition, color, and isVisible. It includes a constructor that initializes these attributes to default values (diameter=68, xPosition=230, yPosition=90, color="blue"). It also includes a makeVisible() method that sets isVisible to true and calls a draw() method.

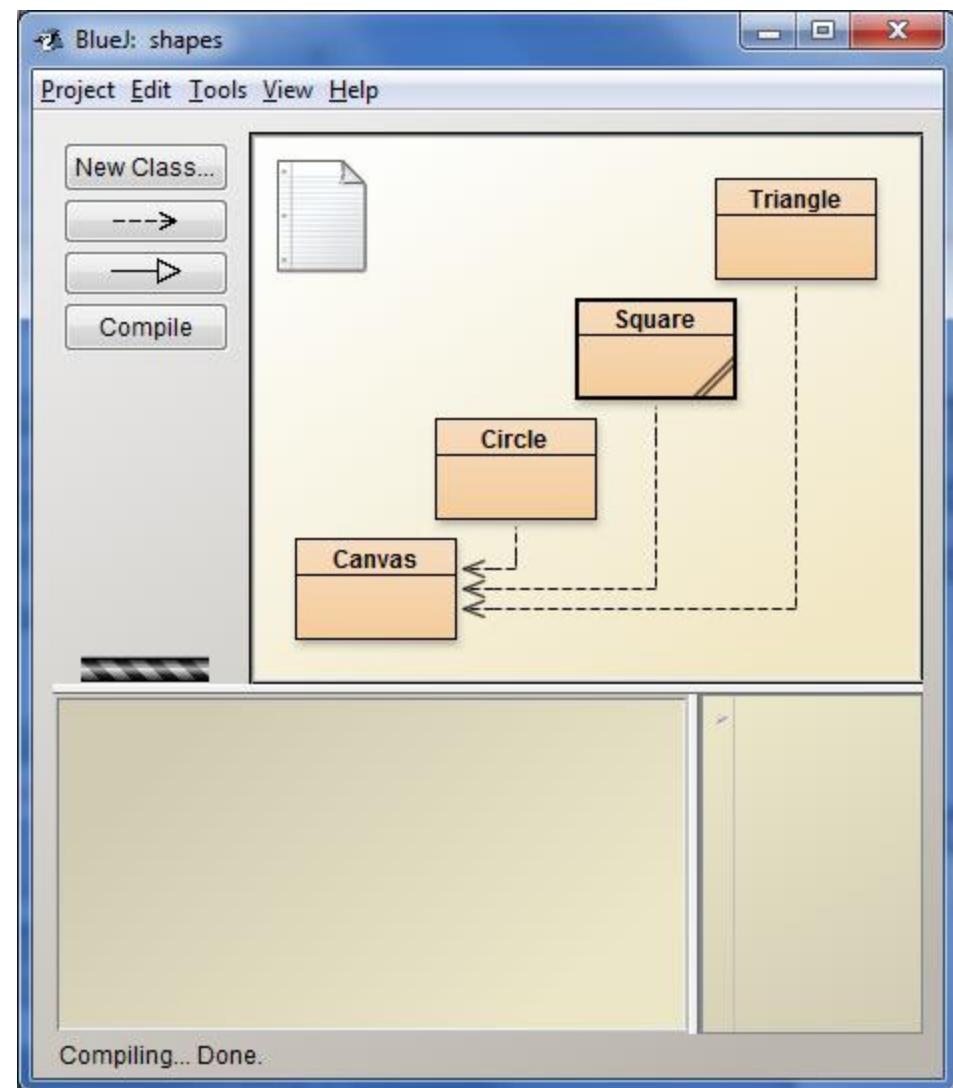
```
/*
 * Circle.java
 */
public class Circle
{
    private int diameter;
    private int xPosition;
    private int yPosition;
    private String color;
    private boolean isVisible;

    /**
     * Create a new circle at default position with default color.
     */
    public Circle()
    {
        diameter = 68;
        xPosition = 230;
        yPosition = 90;
        color = "blue";
    }

    /**
     * Make this circle visible. If it was already visible, do nothing.
     */
    public void makeVisible()
    {
        isVisible = true;
        draw();
    }
}
```

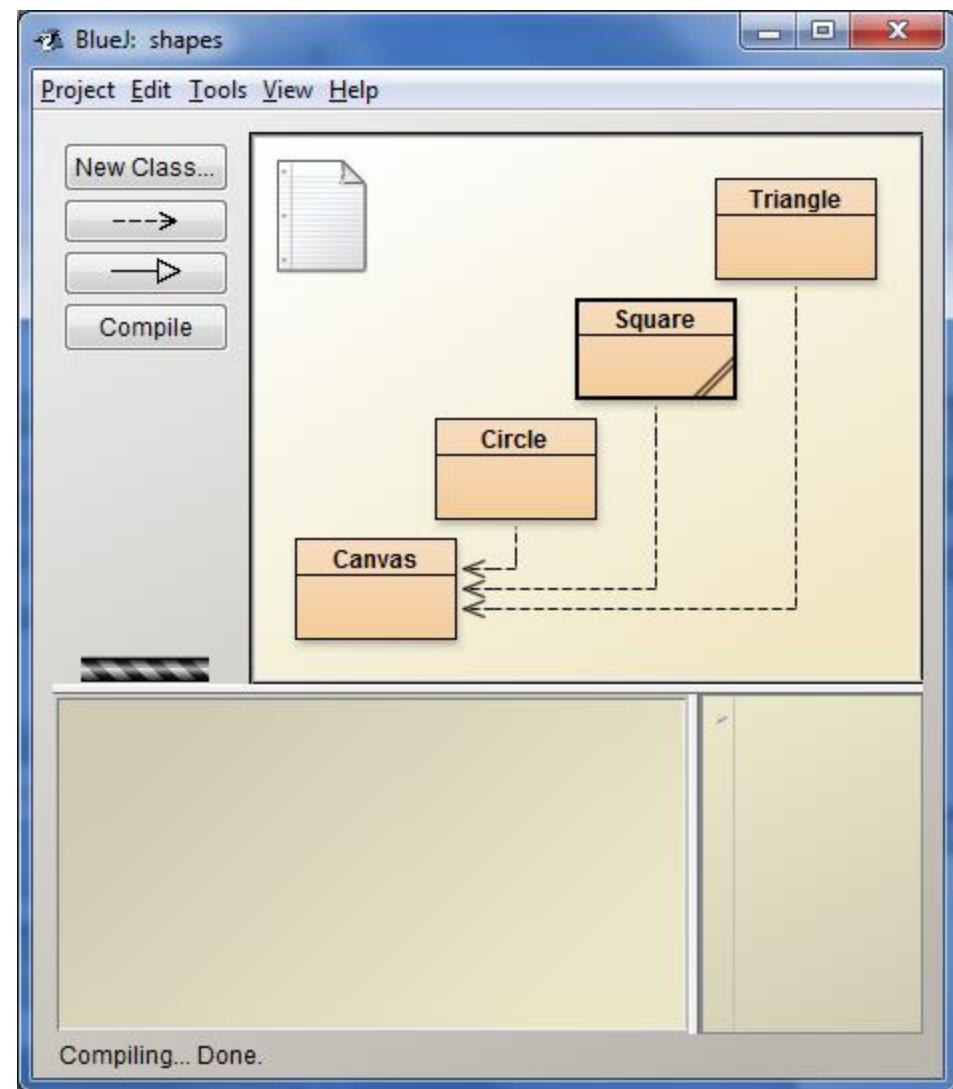
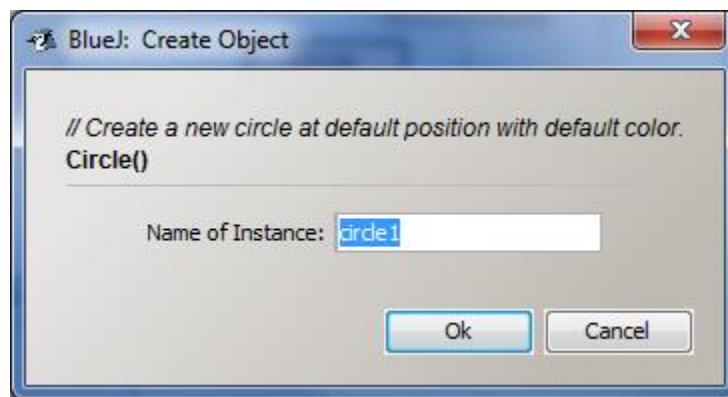
# Objects created appear on the object bench

- To create an instance of class Circle:  
right-click on Circle,  
select *new ...*,  
and choose a name



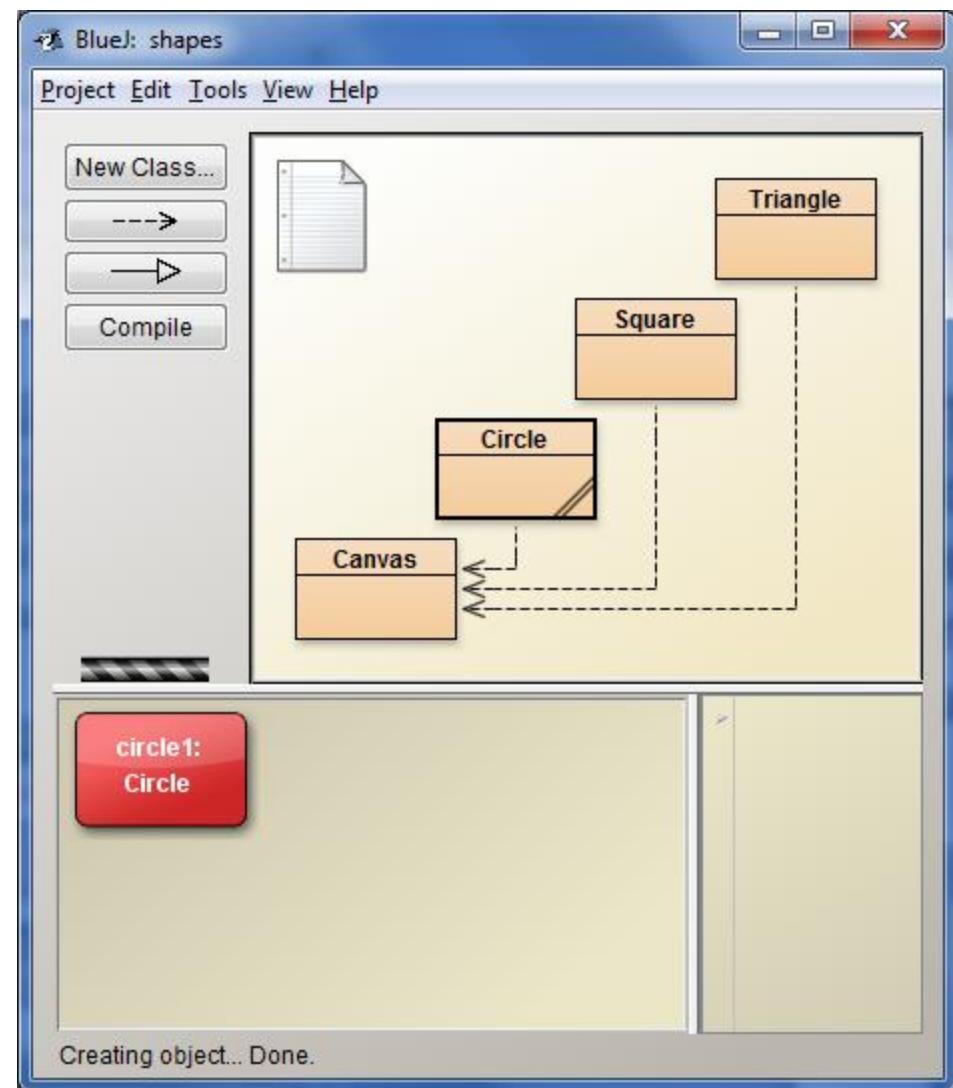
# Objects created appear on the object bench

- To create an instance of class Circle:  
right-click on Circle,  
select *new ...*,  
and choose a name



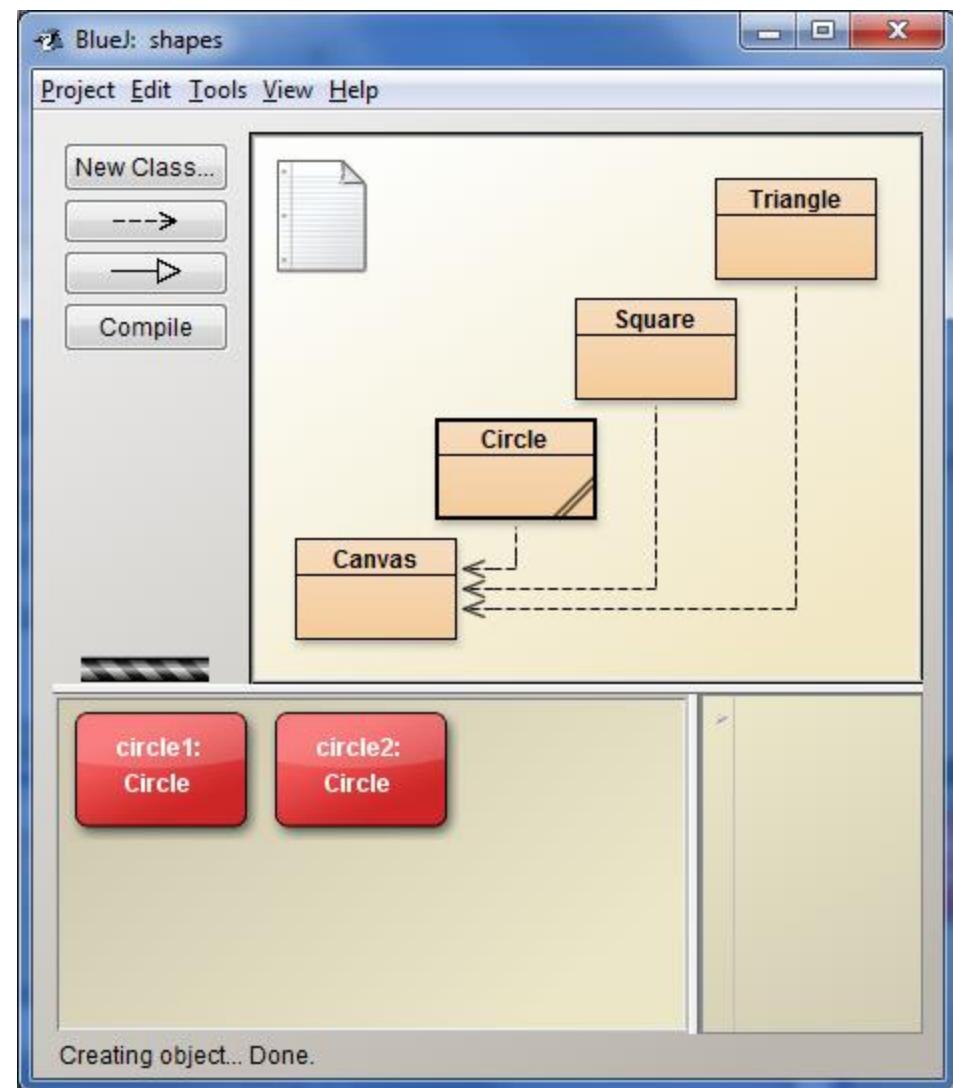
# Objects created appear on the object bench

- To create an instance of class Circle:  
right-click on Circle,  
select *new ...*,  
and choose a name



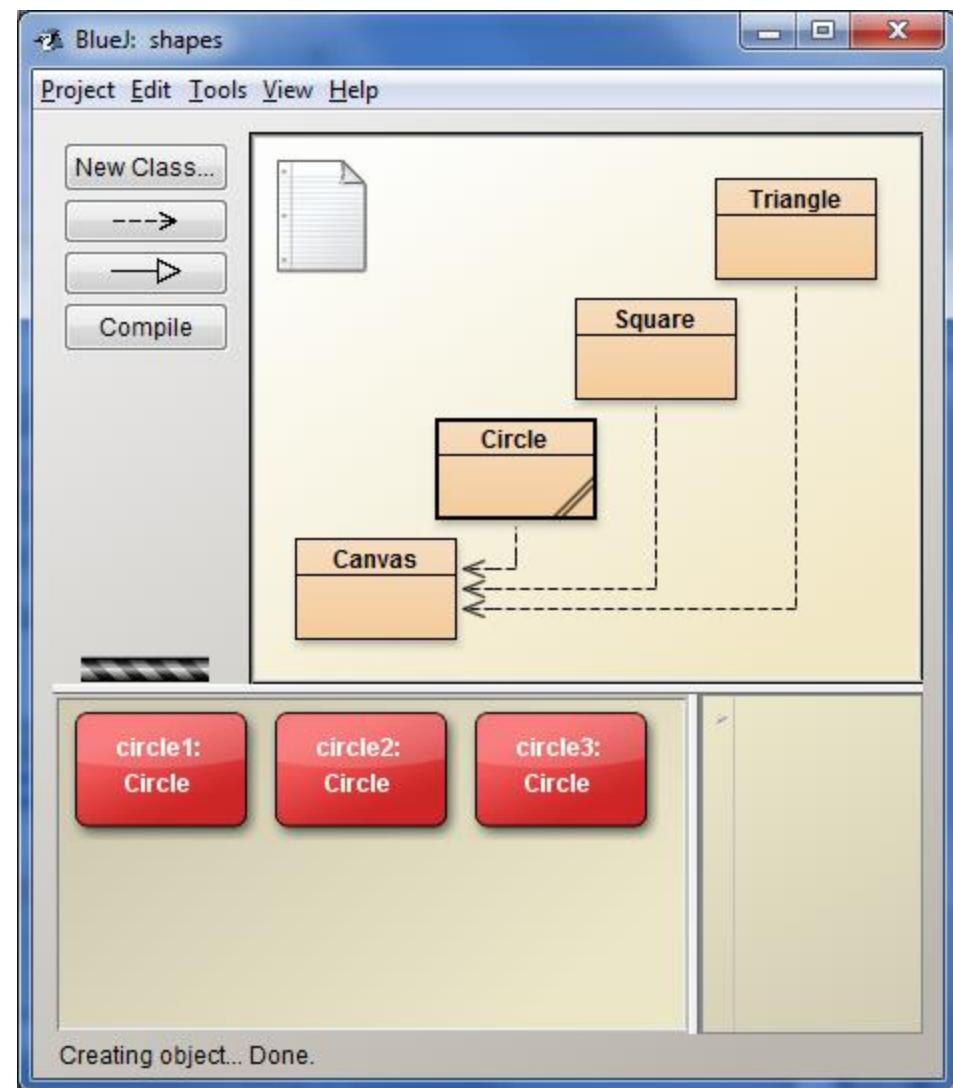
# Objects created appear on the object bench

- To create an instance of class Circle:  
right-click on Circle,  
select *new ...*,  
and choose a name



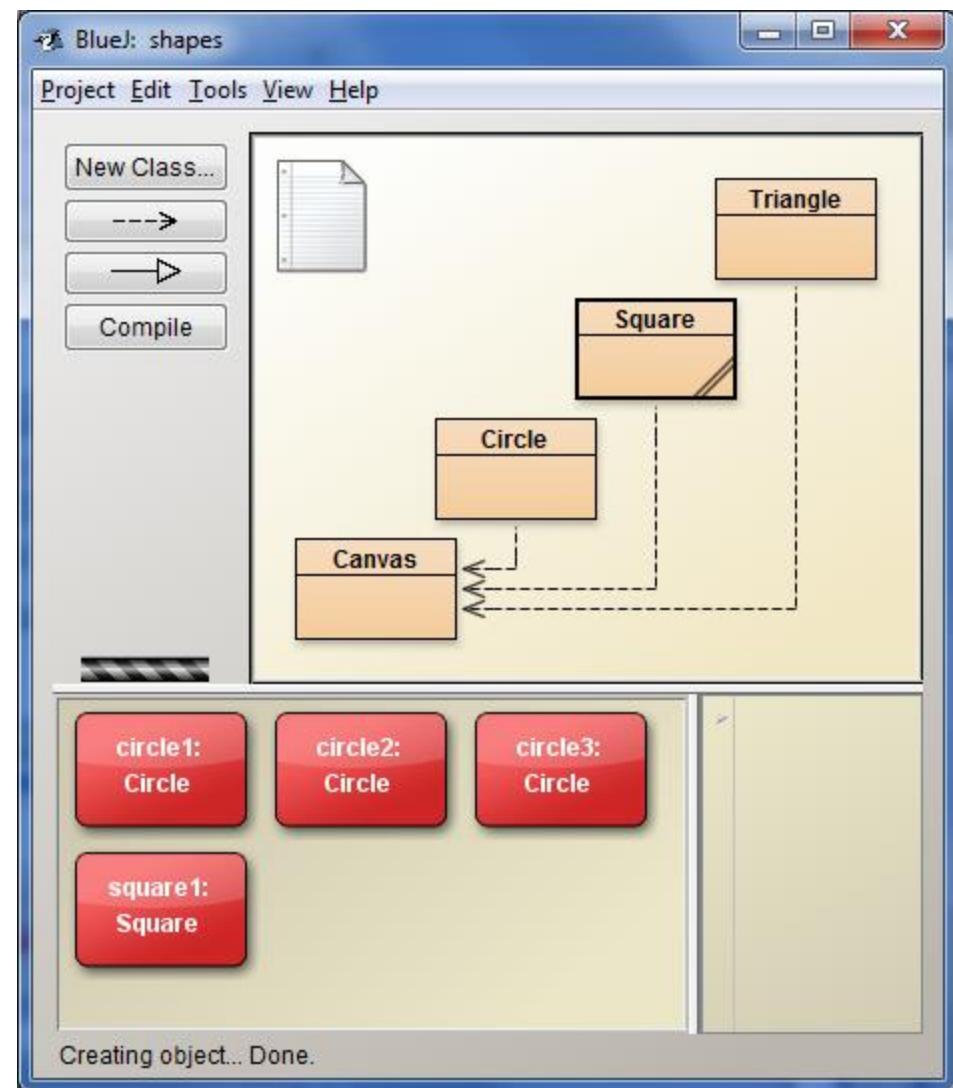
# Objects created appear on the object bench

- To create an instance of class Circle:  
right-click on Circle,  
select *new ...*,  
and choose a name



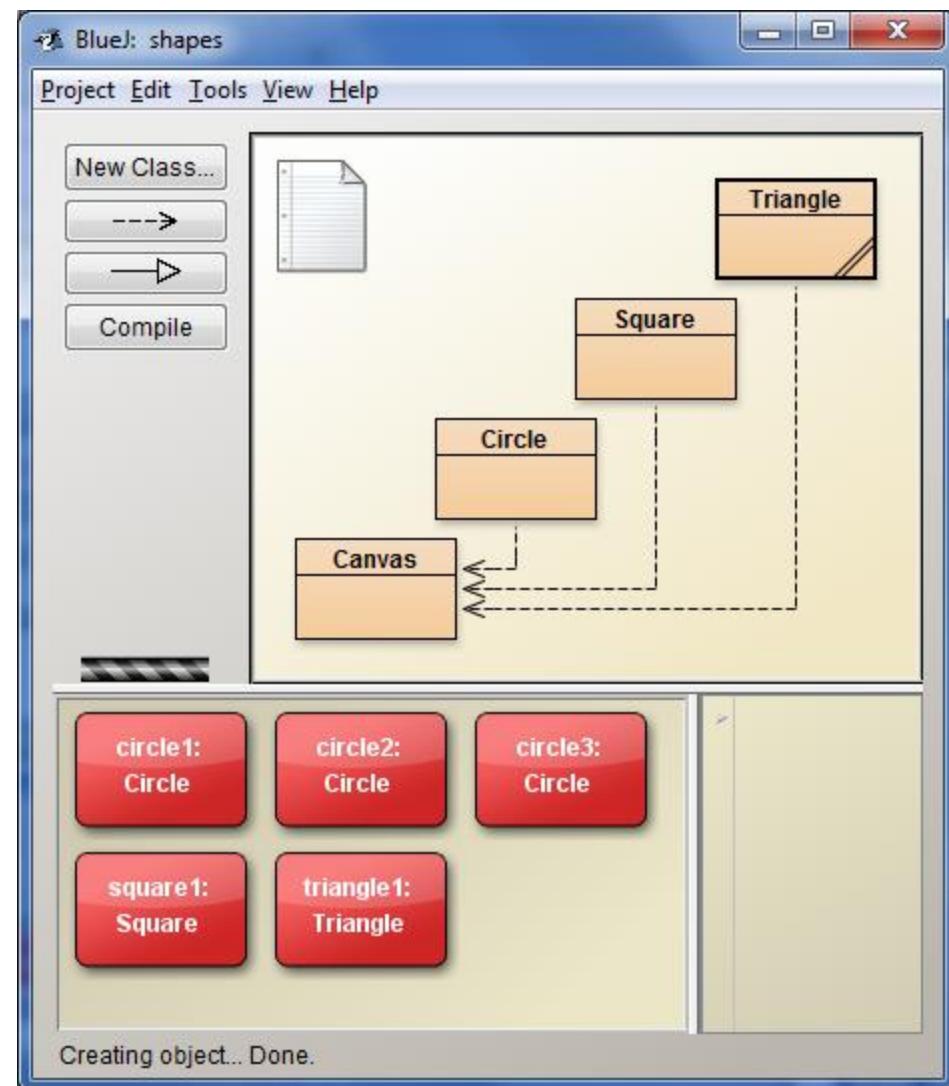
# Objects created appear on the object bench

- To create an instance of class Square:  
right-click on Square,  
select *new ...*,  
and choose a name



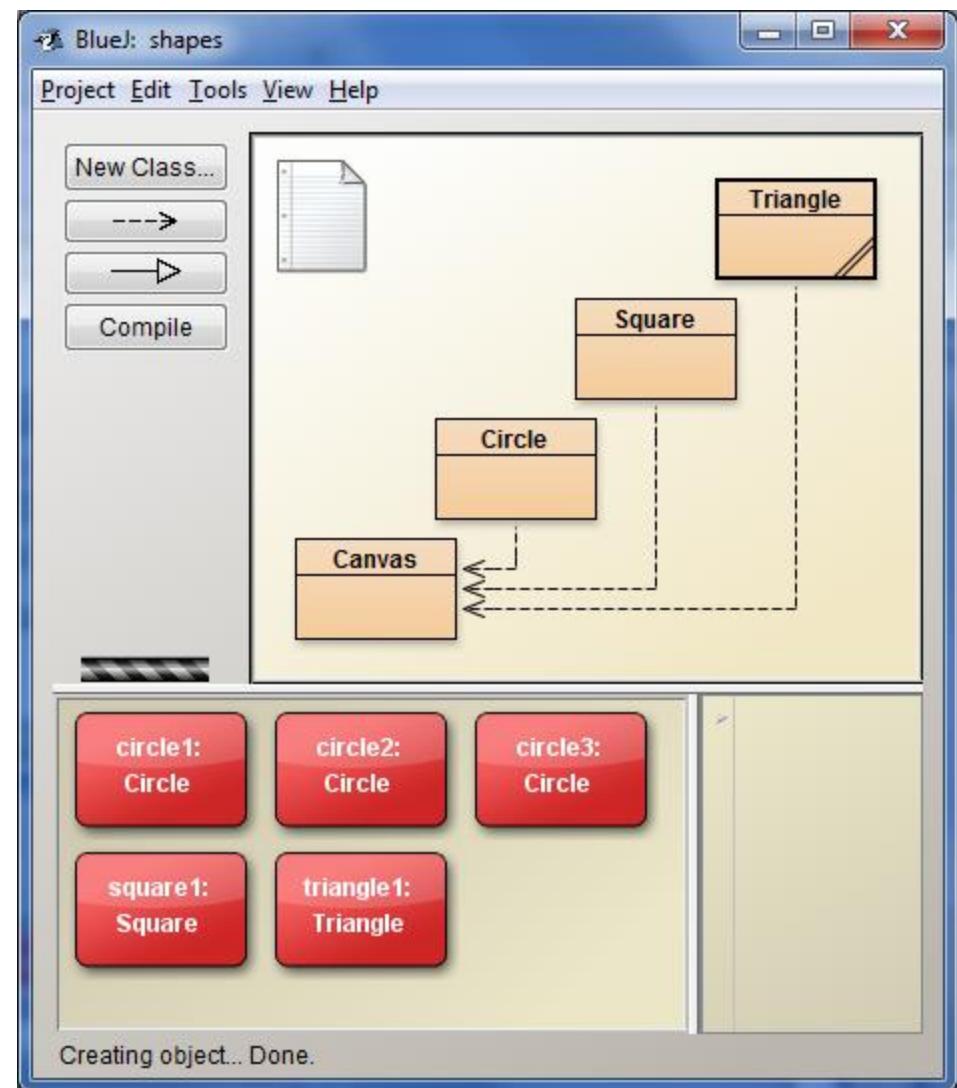
# Objects created appear on the object bench

- To create an instance of class Triangle: right-click on Triangle, select *new ...*, and choose a name



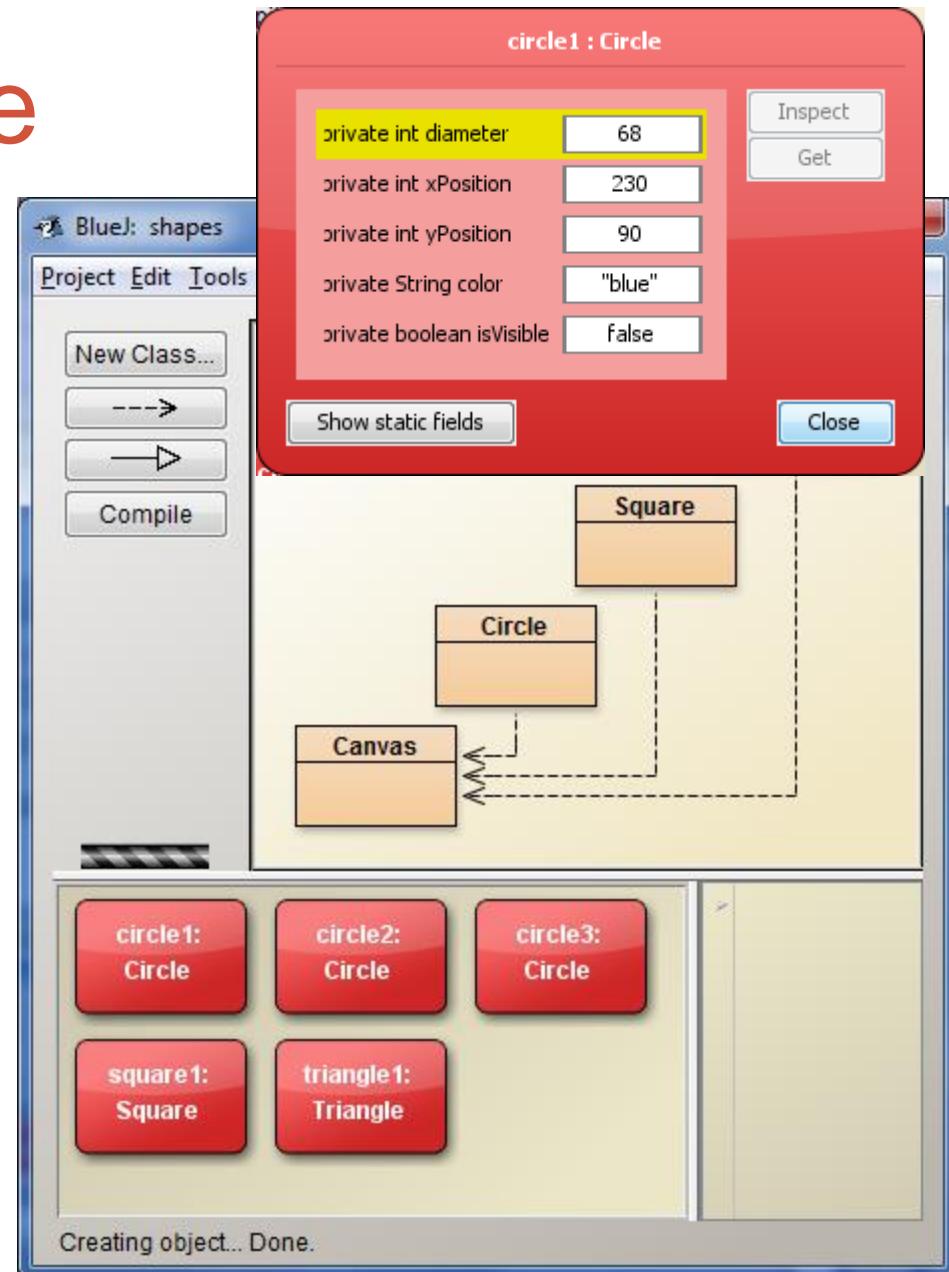
# Objects have state

- To see the state for object X: right-click on X and select *Inspect*
- Each *field* in the state has a type, a name, and a value



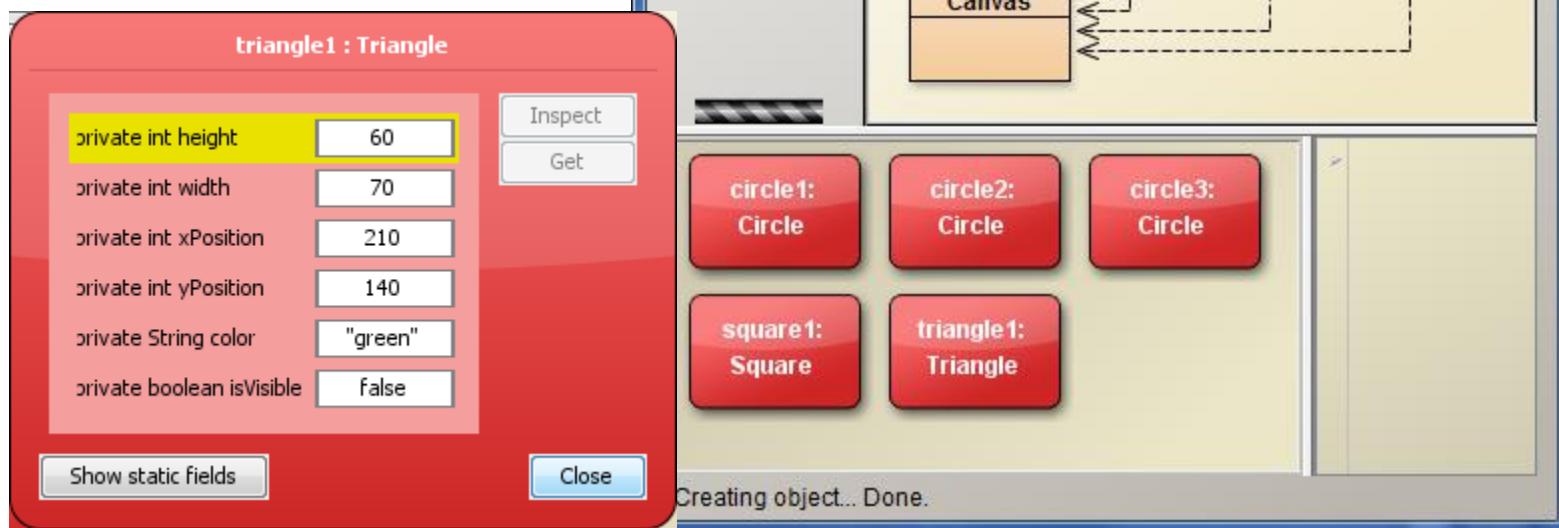
# Objects have state

- To see the state for object X: right-click on X and select *Inspect*
- Each *field* in the state has a type, a name, and a value



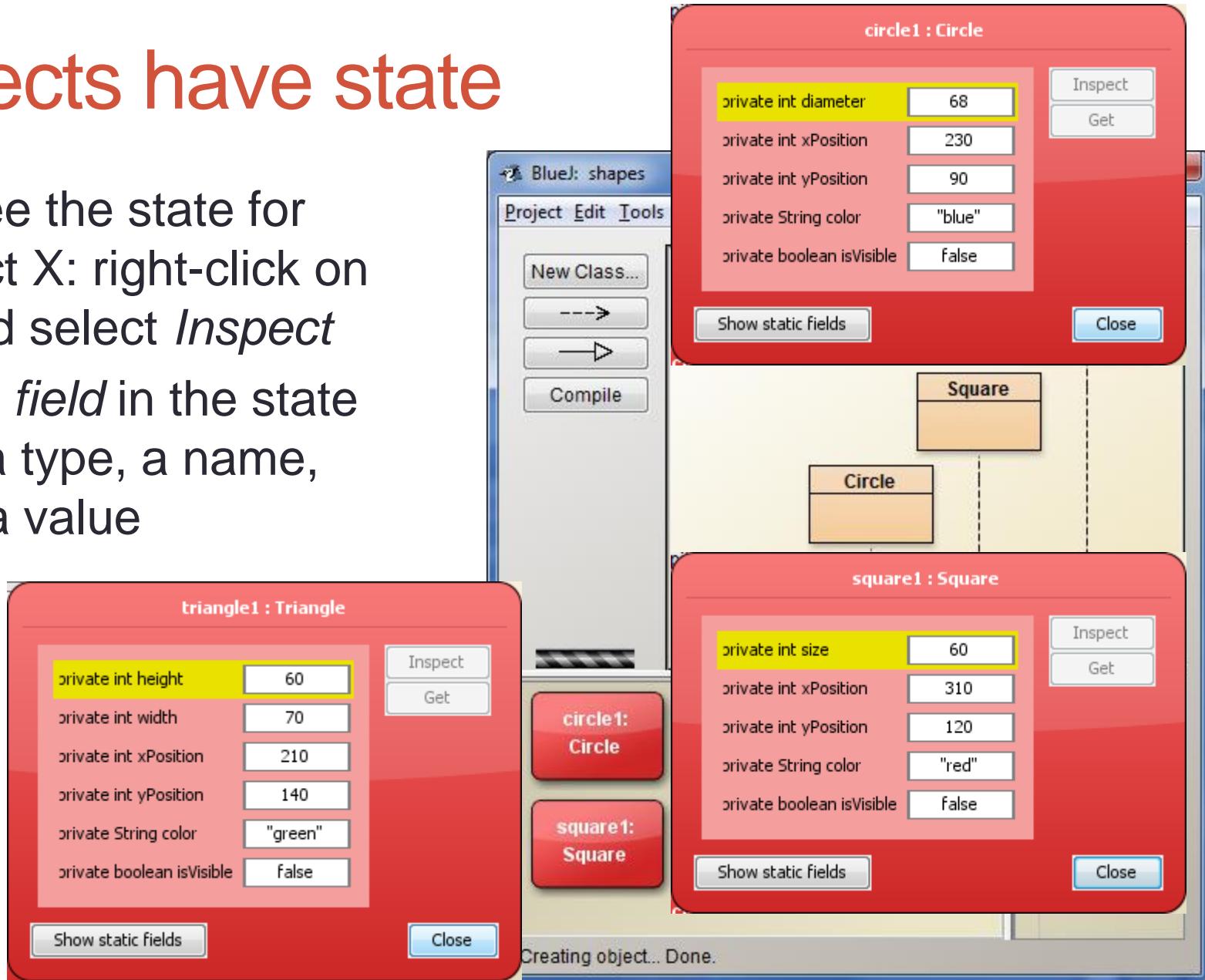
# Objects have state

- To see the state for object X: right-click on X and select *Inspect*
- Each *field* in the state has a type, a name, and a value



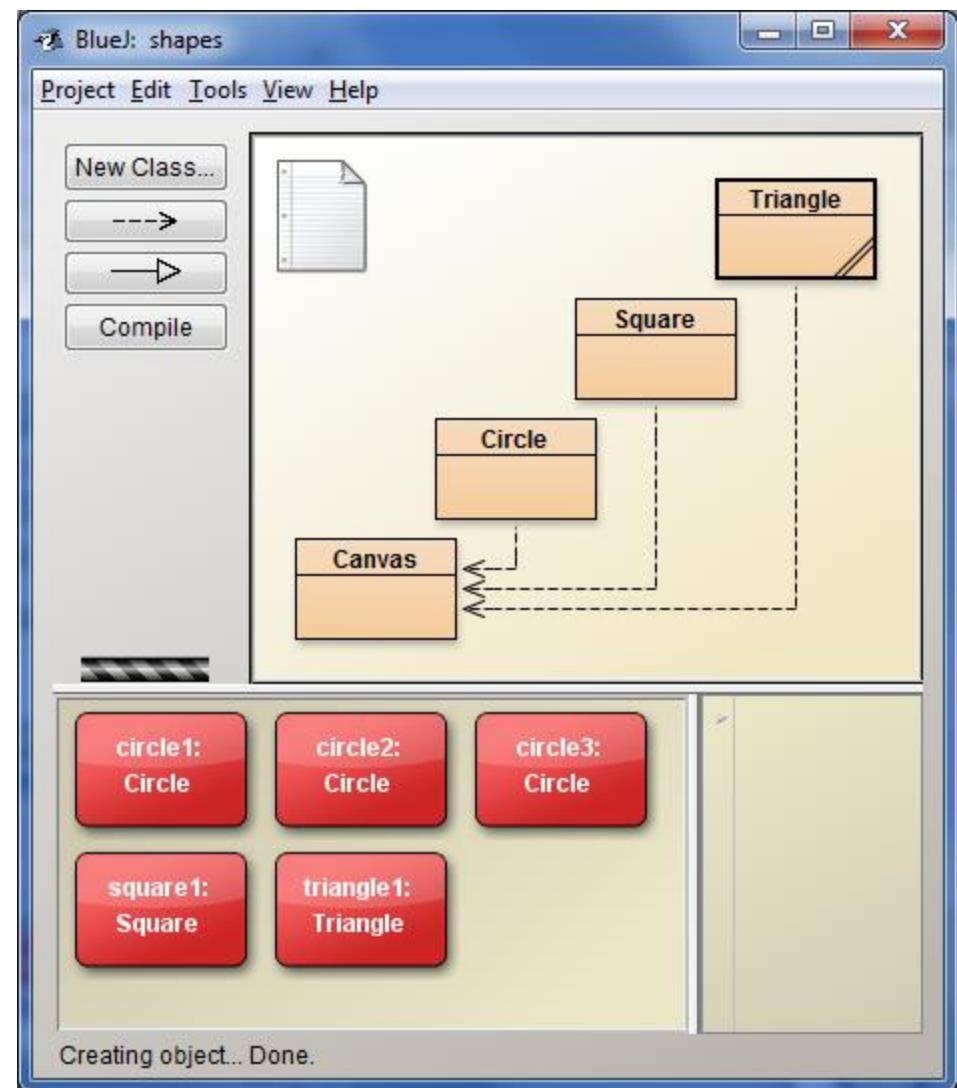
# Objects have state

- To see the state for object X: right-click on X and select *Inspect*
- Each *field* in the state has a type, a name, and a value



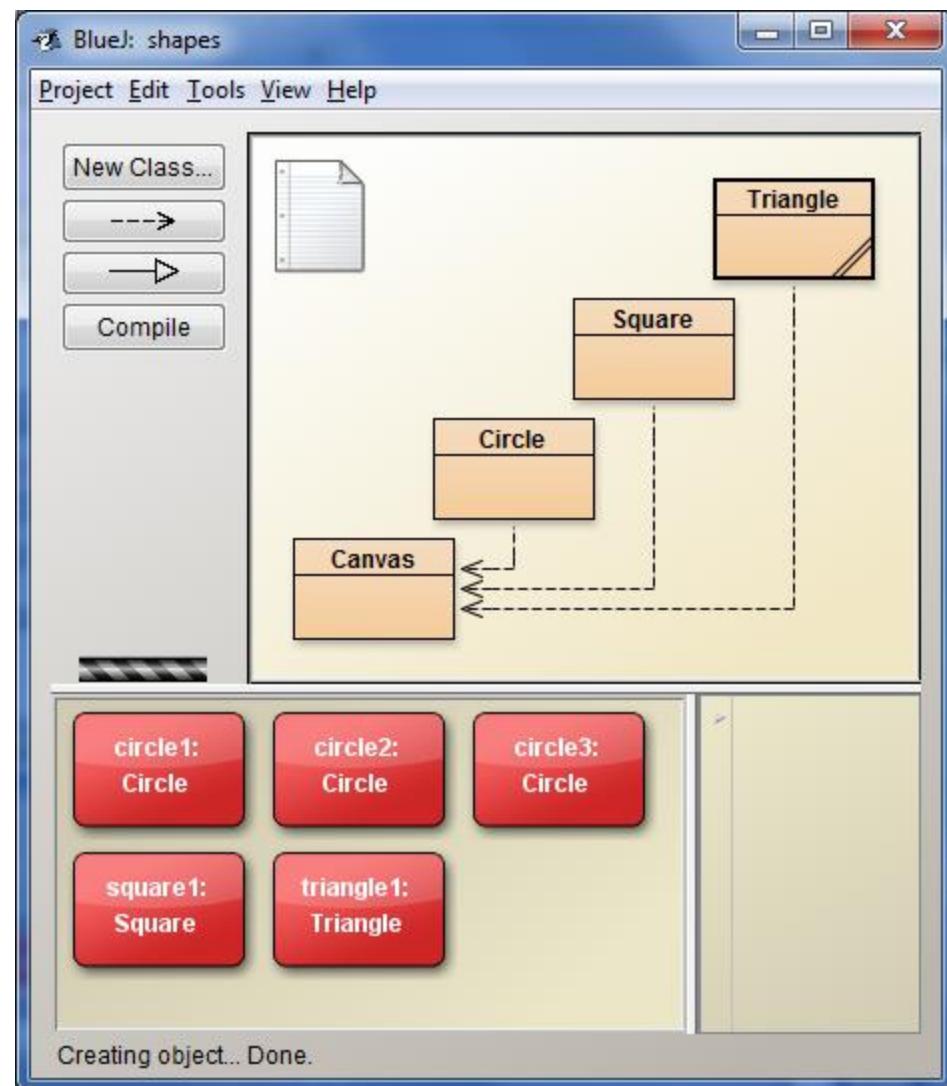
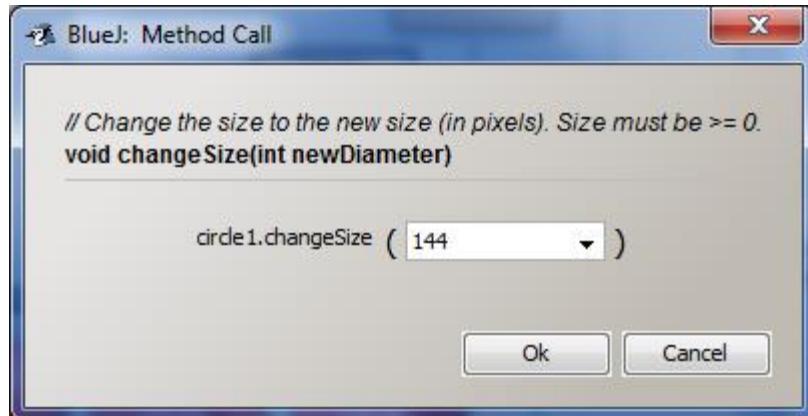
# An object's state can change

- To change the size of circle1: right-click on circle1, select changeSize, and enter a new diameter (an integer)
- Then inspect its state again



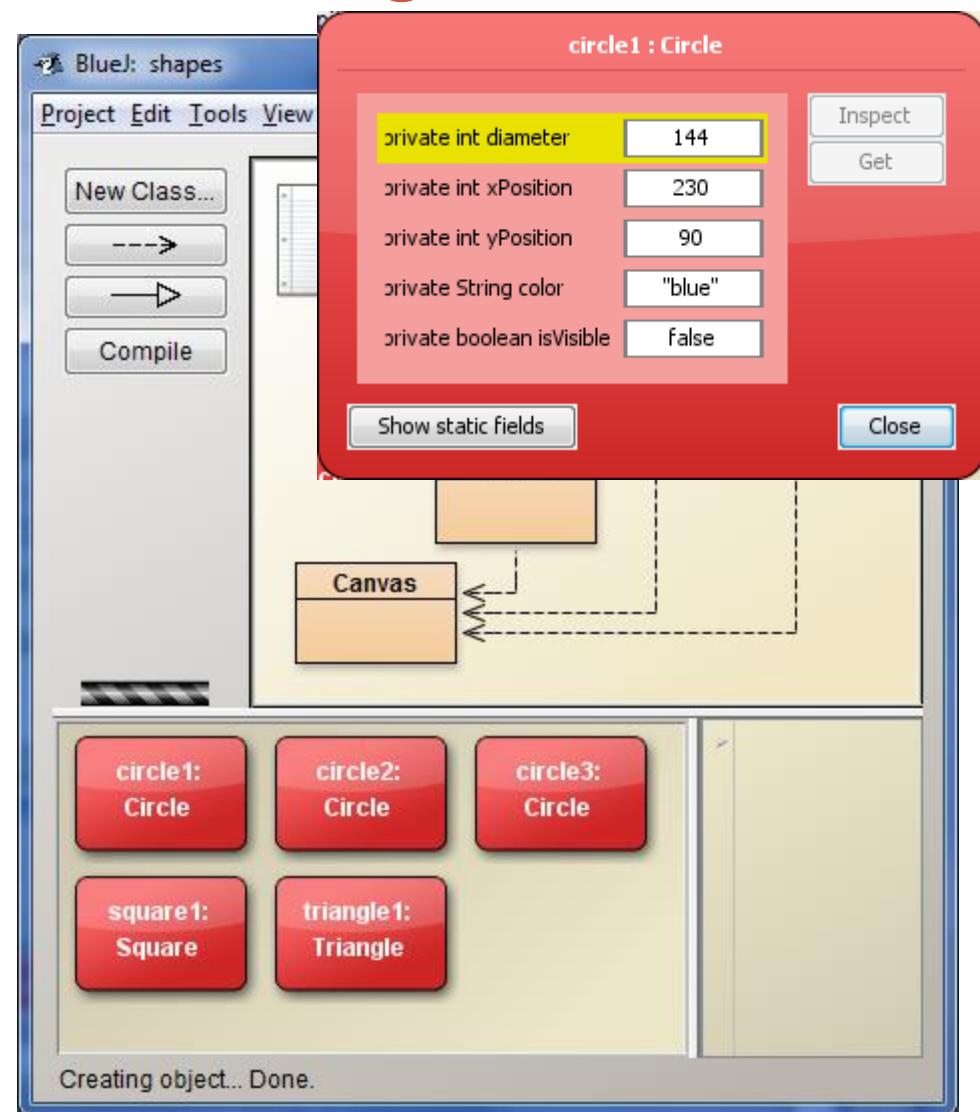
# An object's state can change

- To change the size of circle1: right-click on circle1, select changeSize, and enter a new diameter (an integer)
- Then inspect its state again



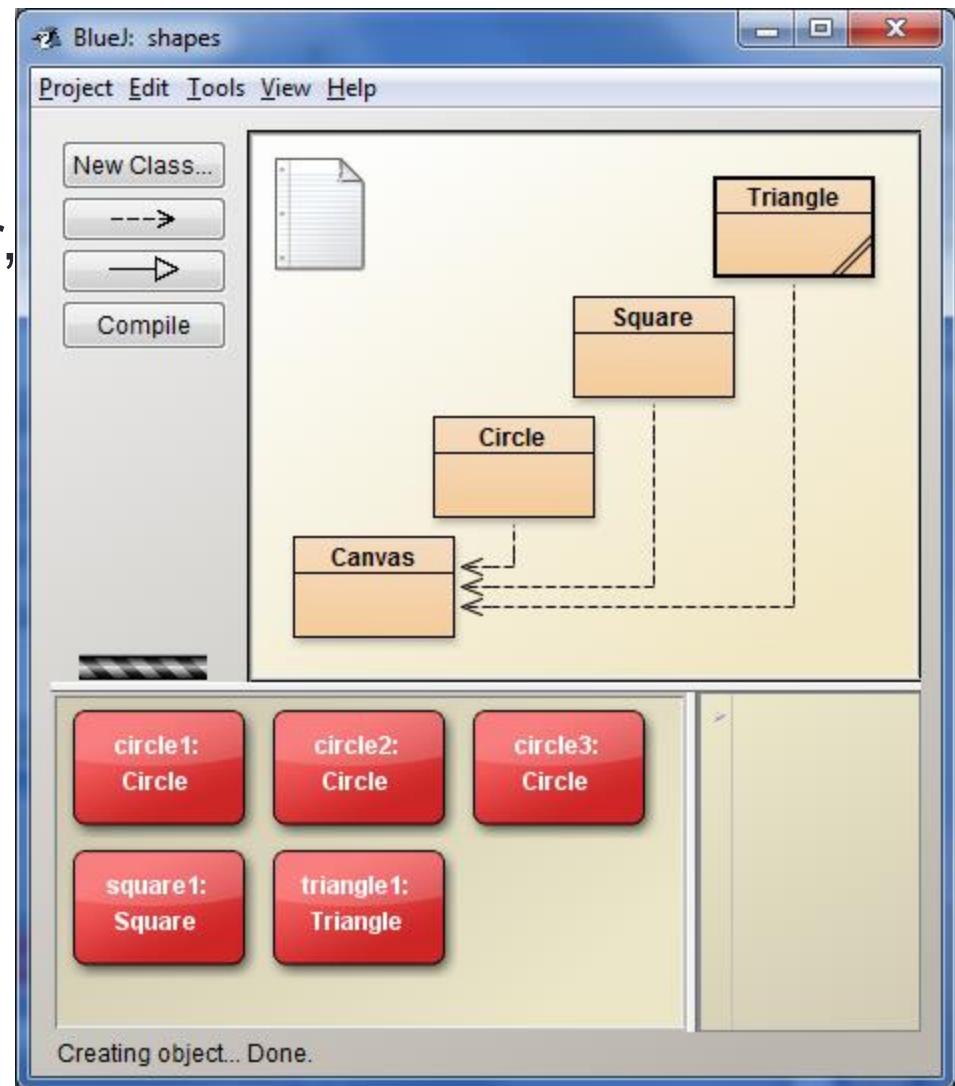
# An object's state can change

- To change the size of circle1: right-click on circle1, select changeSize, and enter a new diameter (an integer)
- Then inspect its state again



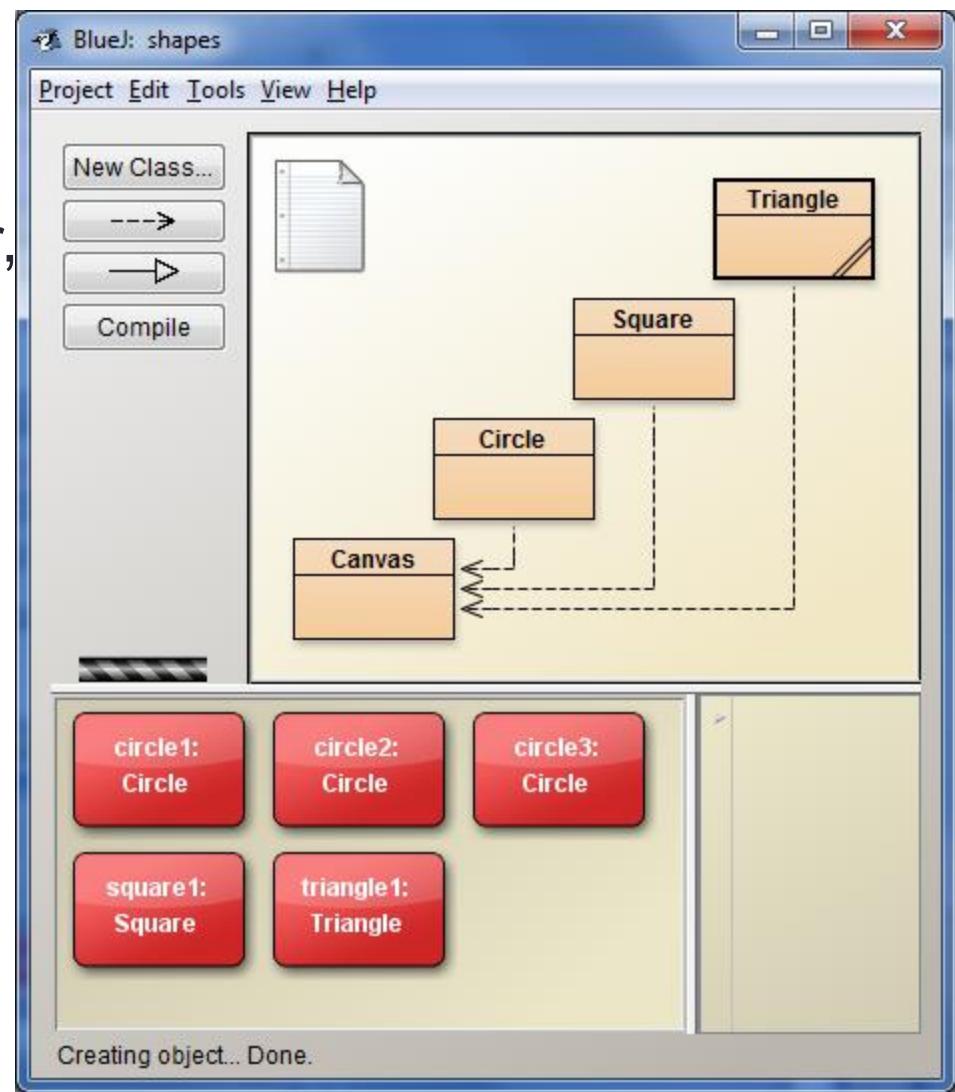
# An object's state can change

- To change the colour of circle1: right-click on circle1, select changeColor, and enter a new colour (a name in double quotes like “red”)
- Then inspect its state again



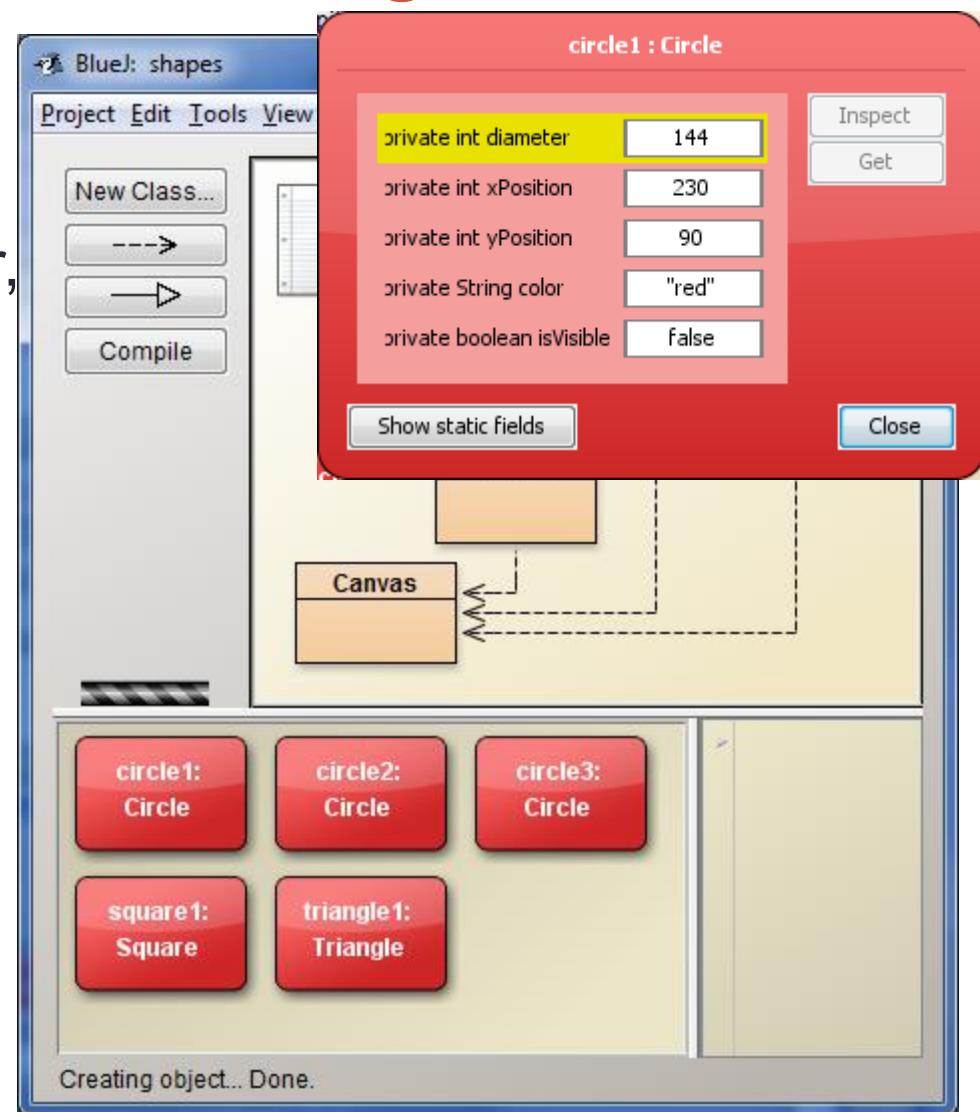
# An object's state can change

- To change the colour of circle1: right-click on circle1, select changeColor, and enter a new colour (a name in double quotes like "red")
- Then inspect its state again



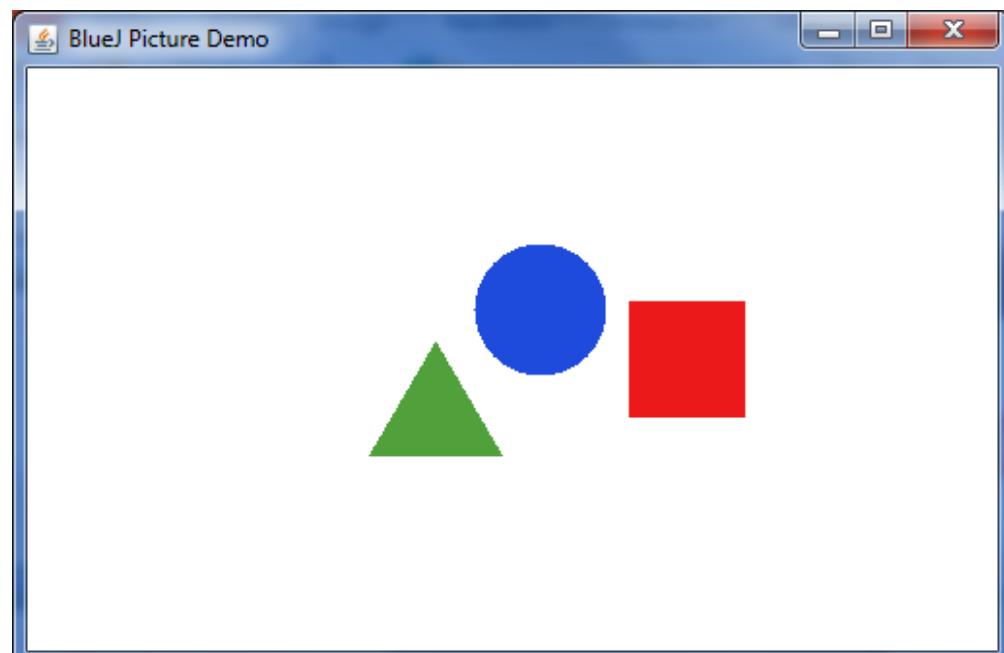
# An object's state can change

- To change the colour of circle1: right-click on circle1, select changeColor, and enter a new colour (a name in double quotes like "red")
- Then inspect its state again



# Objects can do things

- Create a Canvas object (ignore the result returned)
- Make circle2, triangle1, and square1 visible with the appropriate operations
  - And inspect their state again
- Then play!
  - e.g. try the move operations
  - e.g. try making larger pictures



# CITS1001

## 3. CLASS STRUCTURE

---

*Objects First With Java, Chapter 2*

# Lecture essentials

- Class structure in Java
  - Name, fields, constructors, methods
- Introduction to Java syntax
- Introduction to methods
- Code should always be written to be both *machine readable* and *human readable*

# Recap

- Classes are the organisational unit of code in object-oriented languages like Java
- A class takes an entity important to the problem domain and describes
  - The attributes belonging to objects of that type
  - The actions allowed for objects of that type
- The class definition is also used to create objects when the program is executing
  - One class definition can be used to create many objects

# Source code

- A class is defined by a text file of source code
  - With the suffix `.java`
- Source code must specify *everything* about how objects belonging to the class behave
  - Computers are very fast, but also very stupid!
  - The code will do exactly what you *tell* it to do, not what you *meant* it to do
- We will use as a running example `TicketMachine.java`
- Practise reading code, as well as writing code
  - Consider novels, or poetry, or music, or other creative pursuits
  - No one writes these kinds of things without reading other people's efforts first

# Source code

- A class is defined by a text
  - With the suffix `.java`
- Source code must specify the objects belonging to the class
  - Computers are very fast, but also...
  - The code will do exactly what, not what you *meant* it to do
- We will use as a running example
- Practise reading code, as well
  - Consider novels, or poetry, or...
  - No one writes these kinds of things other people's efforts first

## Class TicketMachine – naive-ticket-machine

```

1  /**
2  * TicketMachine models a naive ticket machine that issues
3  * flat-fare tickets.
4  * The price of a ticket is specified via the constructor.
5  * It is a naive machine in the sense that it trusts its users
6  * to insert enough money before trying to print a ticket.
7  * It also assumes that users enter sensible amounts.
8  *
9  * @author David J. Barnes and Michael Kölling
10 * @version 2016.02.29
11 */
12 public class TicketMachine
13 {
14     // The price of a ticket from this machine.
15     private int price;
16     // The amount of money entered by a customer so far.
17     private int balance;
18     // The total amount of money collected by this machine.
19     private int total;
20
21 /**
22 * Create a machine that issues tickets of the given price.
23 * Note that the price must be greater than zero, and there
24 * are no checks to ensure this.
25 */
26 public TicketMachine(int cost)
27 {
28     price = cost;
29     balance = 0;
30     total = 0;
31 }
32
33 /**
34 * Return the price of a ticket.
35 */
36 public int getPrice()
37 {
38     return price;
39 }
40
41 /**
42 * Return the amount of money already inserted for the
43 * next ticket.
44 */
45 public int getBalance()
46 {
47     return balance;
48 }
49

```

# What is a programming language?

- A program for a computer to follow must be expressed completely unambiguously
- There are many different *programming languages* in which programs can be written
- In order to write a working program, you need to learn
  - the *vocabulary* and *syntax* of the language, so you can write statements that make sense
  - how to make sequences of legal statements that do simple tasks
  - how to express what you want the computer to do, in a simple enough way to translate into the programming language
- Similar to learning the *words*, how to form *sentences*, and how to *write a story*, in learning a human language

# Write both for people and for the computer

- Program code is executed by a computer
  - The computer will do exactly what your program tells it to do
  - The rules of the language determine what happens when the program is run
  - Remember **the computer does not know what you intended the program to do!**
- And good code is designed to be human readable, e.g.
  - Familiar words are used for programming constructs (e.g. if, else, while, repeat, for)
  - Indented format is similar to paragraphs and sections in text
  - Meaningful variable names suggest what is intended (e.g. price, mark, studentName)

“Programming can be difficult at first. It is annoying when your program doesn’t work, and you spend ages trying to figure out why. Bugs can seem to come from nowhere, for no reason. **But there is always a logical reason behind a bug.** It is incredibly satisfying when your program does work.”

*Unknown CITS1001 student*

# Ticket machines – an external view

- The external view of a class means considering
  - What objects of the class do
  - How we create and use those objects
- For example, machines accept money and supply tickets at a fixed price
  - How is that price determined?
  - How is ‘money’ ‘entered’ into a machine?
  - How do we check how much money has been entered?
- This is the view relevant to the *user* of a class
  - As you were in the first lab sheet

# Ticket machines – an internal view

- The internal view of a class means also considering
  - How objects store information
  - How objects do things
- Looking inside allows us to determine how behaviour is provided or implemented
  - How is the current balance recorded and updated?
  - How does the machine check for errors?
  - How does it print tickets?
- This is the view relevant to the *writer* of a class
  - Effectively, the “designer” of ticket machines

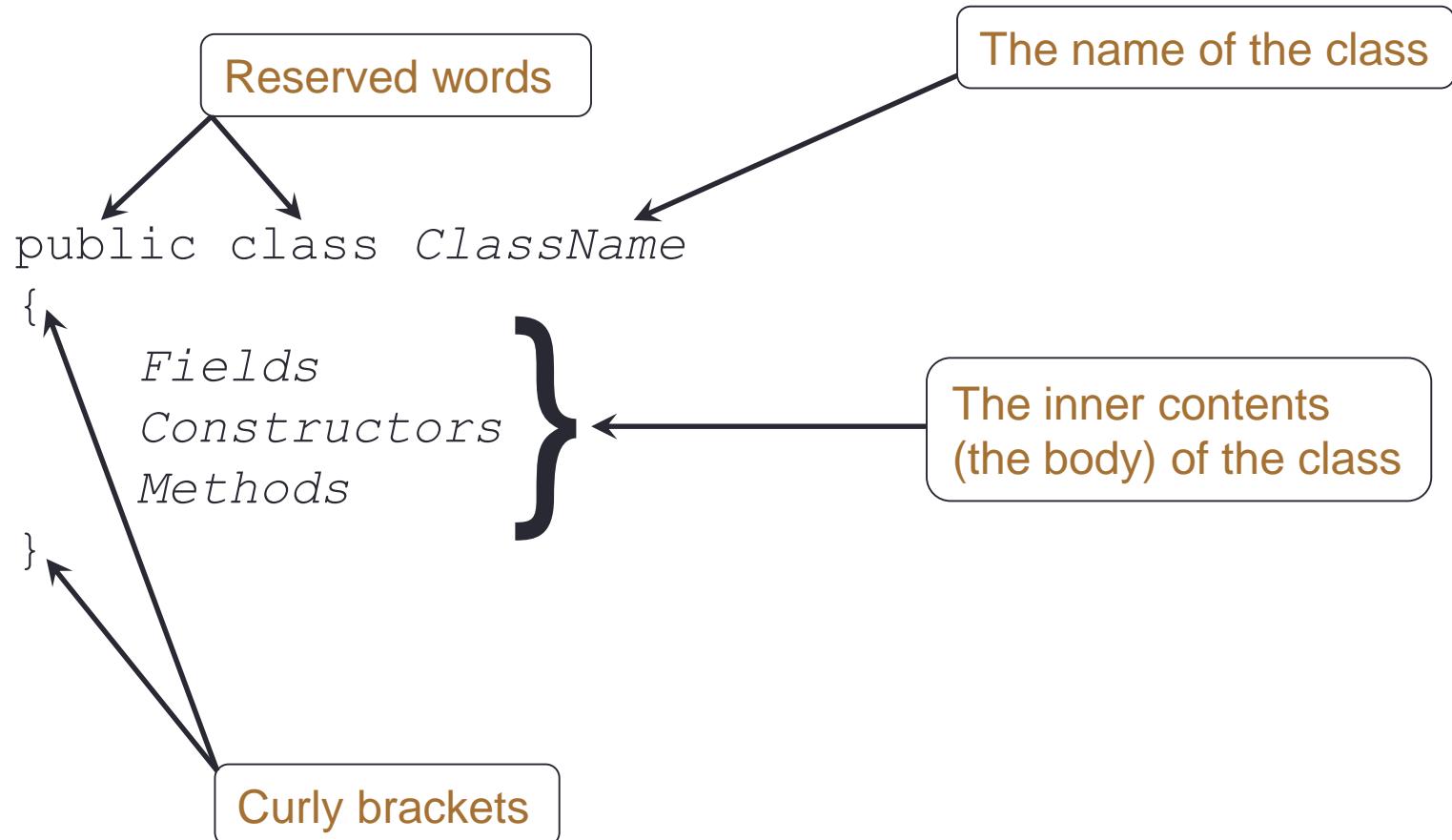
# The four components of a class

- A class definition has four components
  - Its **name** – what is the class called?
  - Its **fields** – what information do we hold for each object, and how is it represented?
  - Its **constructors** – how are objects created?
  - Its **methods** – what can objects do, and how do they do it?
- It is (usually) best to consider the four components in this order, whether you are writing your own class, or reading someone else's
  - But during development, there may be some iteration between defining the various components

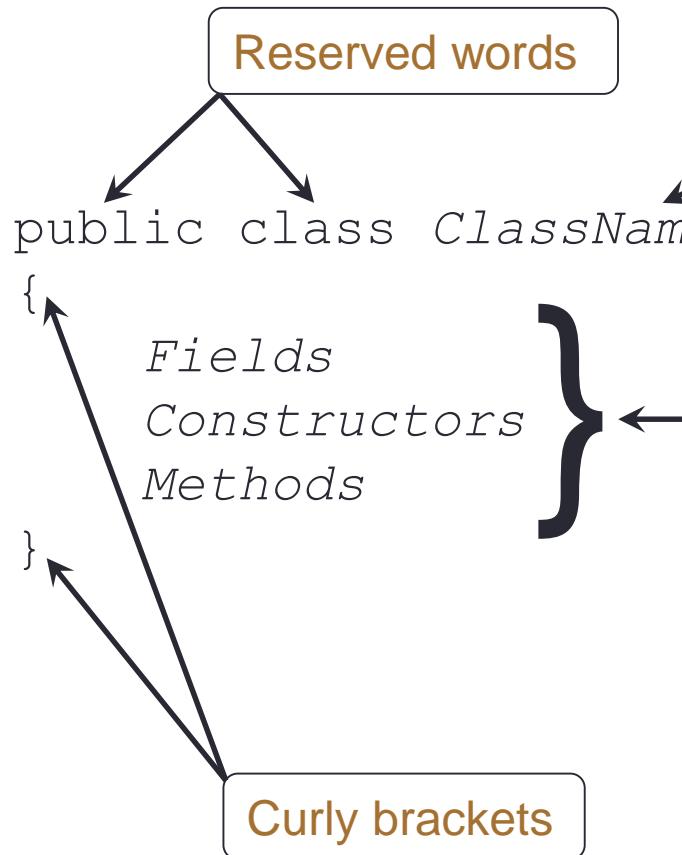
# The lifetime of a ticket machine

- These components mirror the operation of a physical ticket machine in the real world
- The people who write the class are akin to the people who design ticket machines
- Calling the constructor is akin to building and installing a ticket machine at say a train station
  - Note that for most of their lifetime, both real ticket machines and `TicketMachine` objects do nothing!
- Calling a method is like getting the machine to do something
  - e.g. inserting money, enquiring about the current balance, issuing tickets, asking for a refund, etc.
- The fields record the state of the machine, e.g. the balance
  - Determined by the history of its actions

# Basic class syntax



# Basic class syntax



## Class TicketMachine – naive-ticket-machine

1/2

```

1  /**
2  * TicketMachine models a naive ticket machine that issues
3  * flat-fare tickets.
4  * The price of a ticket is specified via the constructor.
5  * It is a naive machine in the sense that it trusts its users
6  * to insert enough money before trying to print a ticket.
7  * It also assumes that users enter sensible amounts.
8  *
9  * @author David J. Barnes and Michael Kölling
10 * @version 2016.02.29
11 */
12 public class TicketMachine
13 {
14     // The price of a ticket from this machine.
15     private int price;
16     // The amount of money entered by a customer so far.
17     private int balance;
18     // The total amount of money collected by this machine.
19     private int total;
20
21     /**
22      * Create a machine that issues tickets of the given price.
23      * Note that the price must be greater than zero, and there
24      * are no checks to ensure this.
25      */
26     public TicketMachine(int cost)
27     {
28         price = cost;
29         balance = 0;
30         total = 0;
31     }
32
33     /**
34      * Return the price of a ticket.
35      */
36     public int getPrice()
37     {
38         return price;
39     }
40
41     /**
42      * Return the amount of money already inserted for the
43      * next ticket.
44      */
45     public int getBalance()
46     {
47         return balance;
48     }
49

```

# Syntax

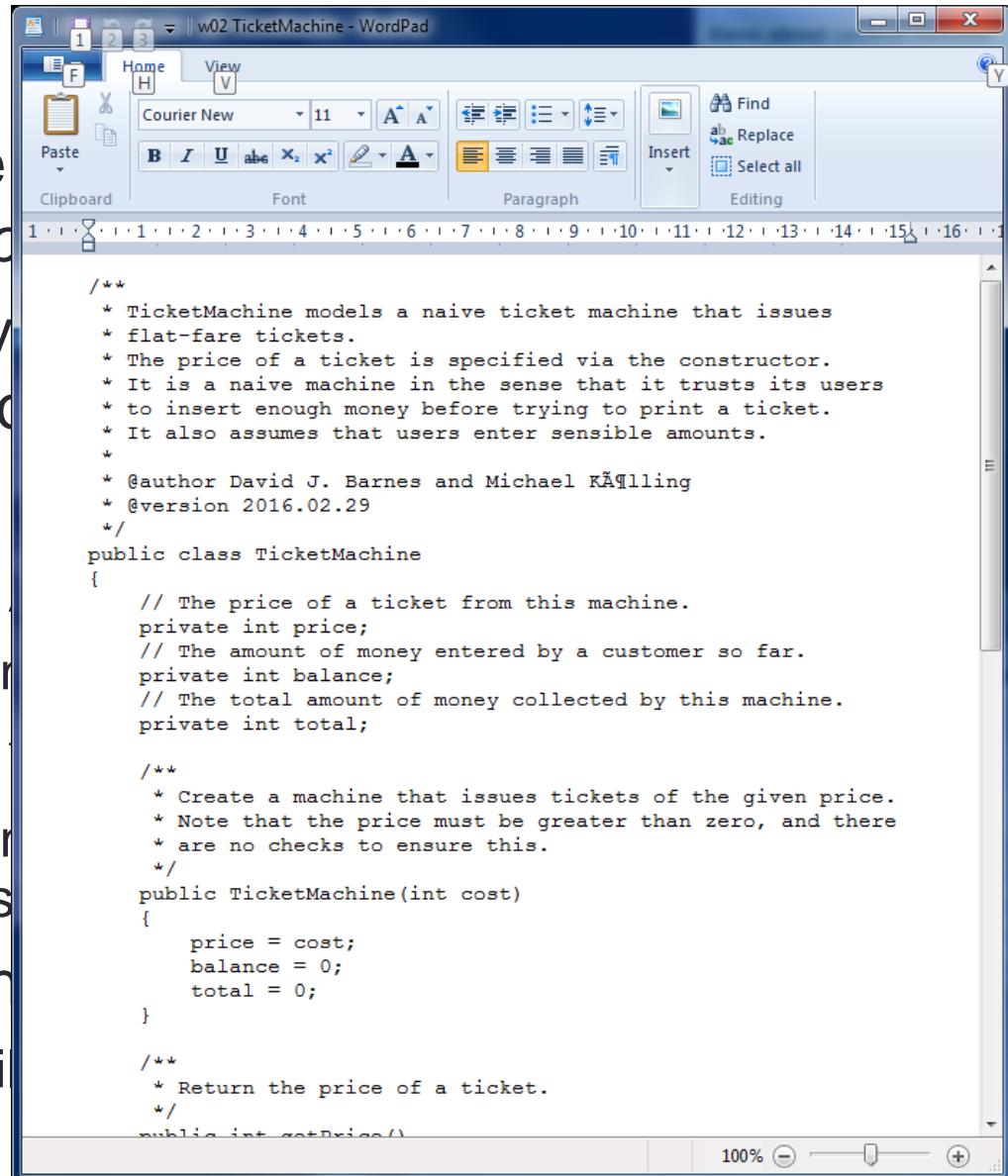
- Reserved words and curly brackets are our first encounter with *Java syntax*
- Source code must be structured in a certain way, as determined by the rules of the language
- Reserved words are words with a special meaning in Java
  - e.g. public, class, private, int, String
  - There are many, many others that we will meet as we go along
  - Also known as *keywords*
- Brackets (of all types) are everywhere in many languages
  - Basically used to group things together
  - Here, the curly brackets delimit the body of the given class

# Comments

- Another thing you will see in the source file *TicketMachine.java* are *comments*
- Comments are ignored by the computer; they exist simply to make the code easier for people to understand
- Comments come in three principal types
- Comments starting with `//`
  - In this case, the computer ignores everything up to the end of the line
- Comments starting with `/*`
  - In this case, the computer ignores everything up to the first occurrence of `*/`, which acts like a closing bracket for the comment
- *Javadoc comments* starting with `/**`
  - Same closing bracket; we will discuss these later in the unit

# Comments

- Another thing you will see in *TicketMachine.java* are comments.
- Comments are ignored by the computer to make the code easier for us to read.
- Comments come in three forms:
- Comments starting with `/*`:
  - In this case, the computer ignores everything until the next occurrence of `*/`, which acts as a closing bracket.
- Comments starting with `/**`:
- *Javadoc comments* starting with `/**`:
  - Same closing bracket; we will learn about these later.



The screenshot shows a Microsoft WordPad window titled "w02 TicketMachine - WordPad". The window displays Java code for a class named `TicketMachine`. The code includes Javadoc-style multi-line comments at the top, constructor code, and a method for getting the ticket price. The WordPad interface is visible, including the ribbon bar with tabs like Home, View, Insert, Find, Replace, and Select all, along with various font and paragraph formatting tools.

```
/**  
 * 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 2016.02.29  
 */  
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 cost)  
    {  
        price = cost;  
        balance = 0;  
        total = 0;  
    }  
  
    /**  
     * Return the price of a ticket.  
     */  
    public int getPrice()  
}
```

## Class TicketMachine – naive-ticket-machine

1/2

# Comments

- Another thing you will see in *TicketMachine.java* are *comments*
- Comments are ignored by the computer to make the code easier to read
- Comments come in three parts
- Comments starting with `/*`
  - In this case, the computer ignores all occurrence of `*/`, which acts like a closing bracket
- Comments starting with `/**`
- Javadoc comments starting with `/**`
  - Same closing bracket; we will learn more about these later

```

1  /**
2  * TicketMachine models a naive ticket machine that issues
3  * flat-fare tickets.
4  * The price of a ticket is specified via the constructor.
5  * It is a naive machine in the sense that it trusts its users
6  * to insert enough money before trying to print a ticket.
7  * It also assumes that users enter sensible amounts.
8  *
9  * @author David J. Barnes and Michael Kölling
10 * @version 2016.02.29
11 */
12 public class TicketMachine
13 {
14     // The price of a ticket from this machine.
15     private int price;
16     // The amount of money entered by a customer so far.
17     private int balance;
18     // The total amount of money collected by this machine.
19     private int total;
20
21 /**
22 * Create a machine that issues tickets of the given price.
23 * Note that the price must be greater than zero, and there
24 * are no checks to ensure this.
25 */
26 public TicketMachine(int cost)
27 {
28     price = cost;
29     balance = 0;
30     total = 0;
31 }
32
33 /**
34 * Return the price of a ticket.
35 */
36 public int getPrice()
37 {
38     return price;
39 }
40
41 /**
42 * Return the amount of money already inserted for the
43 * next ticket.
44 */
45 public int getBalance()
46 {
47     return balance;
48 }
49

```

# Class name

- The name of a class comes in the class *header*
  - Basically the first line, before the opening curly bracket

```
public class TicketMachine
```

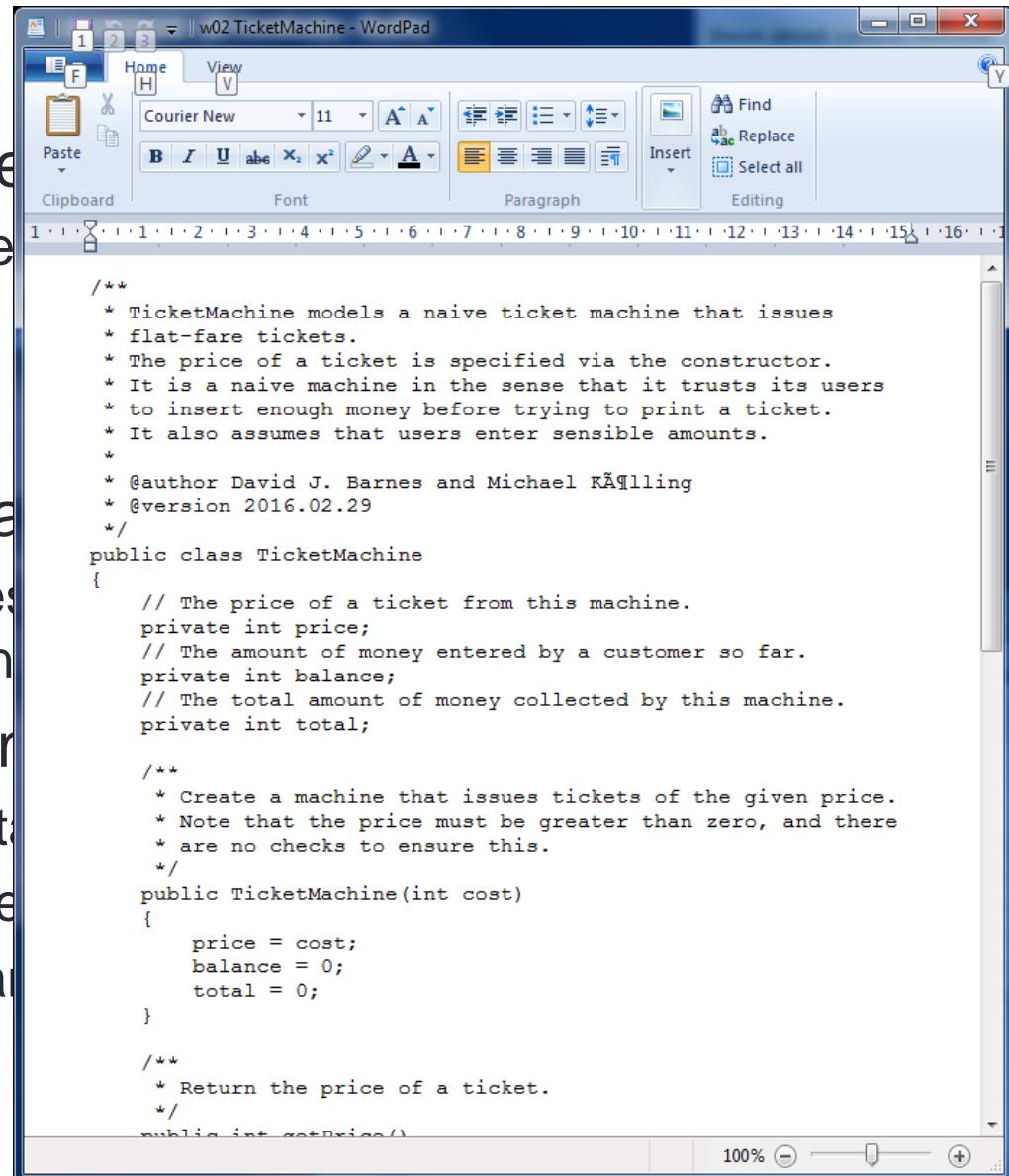
- Make the class name meaningful for readers of the code
  - e.g. the example class represents ticket machines, so we call it *TicketMachine*, not something meaningless like *Xxyz*
- The three rules for the name are that
  - It normally starts with a capital letter
  - It normally contains only letters, digits, and/or underscore, dollar, etc.
  - It must not be the same as any reserved word in the language

# Class name

- The name of a class comes from:
  - Basically the first line, before the opening brace:

```
public class
```

- Make the class name meaningful:
  - e.g. the example class represents a ticket machine called *TicketMachine*, not something else.
- The three rules for the name:
  - It normally starts with a capital letter.
  - It normally contains only letters, numbers, and underscores.
  - It must not be the same as another class name.



A screenshot of the Microsoft WordPad application window titled "w02 TicketMachine - WordPad". The window has a standard Windows-style title bar and menu bar. The main area contains Java code for a class named `TicketMachine`. The code includes a multi-line comment at the top describing the class as a naive ticket machine that issues flat-fare tickets, its constructor parameters, and its methods. Below this is the class definition with private fields `price`, `balance`, and `total`, and their corresponding constructor and getter method. The code uses standard Java syntax with annotations like `/**` for documentation and `private int` for field declarations.

```
/*
 * 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 Käfling
 * @version 2016.02.29
 */
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 cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

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

# Fields/attributes

- The state of an object is a collection of *fields* or *attributes*
- The source code specifies the collection of attributes that each object in the class has
- Each object belonging to a given class has the same attributes, but the *values* of those attributes may be different for different objects
  - e.g. every student has a student number, but different students have different student numbers
  - e.g. every student has a mark, but different students can have different marks
- Note that some fields will change their value often, others less so or even never
  - e.g. your student number never changes, but your marks do!

# The fields of *TicketMachine*

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

**NOTE THE SEMI-COLONS!**

- Each field is described by a **variable**, which has
  - A *visibility modifier*, which denotes who can access it (more later)
  - A *type*, which denotes what values it can store (more later)
  - A *name*, chosen to make its use clear to human readers
- Additionally, and crucially, each field has a **meaning**
  - A sense of what information it stores
  - See *TicketMachine.java* for examples
  - This should apply to every variable in every program you ever write

## Class TicketMachine – naive-ticket-machine

1/2

# The fields of *TicketMachine*

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

- Each field is described by a triple:
  - A *visibility modifier*, which denotes who can see it.
  - A *type*, which denotes what values it can have.
  - A *name*, chosen to make its use clear.
- Additionally, and crucially, each field is annotated with:
  - A sense of what information it represents.
  - See *TicketMachine.java* for examples.
  - This should apply to every variable in the class.

```

1  /**
2  * TicketMachine models a naive ticket machine that issues
3  * flat-fare tickets.
4  * The price of a ticket is specified via the constructor.
5  * It is a naive machine in the sense that it trusts its users
6  * to insert enough money before trying to print a ticket.
7  * It also assumes that users enter sensible amounts.
8  *
9  * @author David J. Barnes and Michael Kölling
10 * @version 2016.02.29
11 */
12 public class TicketMachine
13 {
14     // The price of a ticket from this machine.
15     private int price;
16     // The amount of money entered by a customer so far.
17     private int balance;
18     // The total amount of money collected by this machine.
19     private int total;
20
21     /**
22      * Create a machine that issues tickets of the given price.
23      * Note that the price must be greater than zero, and there
24      * are no checks to ensure this.
25      */
26     public TicketMachine(int cost)
27     {
28         price = cost;
29         balance = 0;
30         total = 0;
31     }
32
33     /**
34      * Return the price of a ticket.
35      */
36     public int getPrice()
37     {
38         return price;
39     }
40
41     /**
42      * Return the amount of money already inserted for the
43      * next ticket.
44      */
45     public int getBalance()
46     {
47         return balance;
48     }
49
50 }
```

# Constructors

- Source code specifies how objects are *created*

```
public TicketMachine (int cost)
{
    price = cost;
    balance = 0;
    total = 0;
}
```

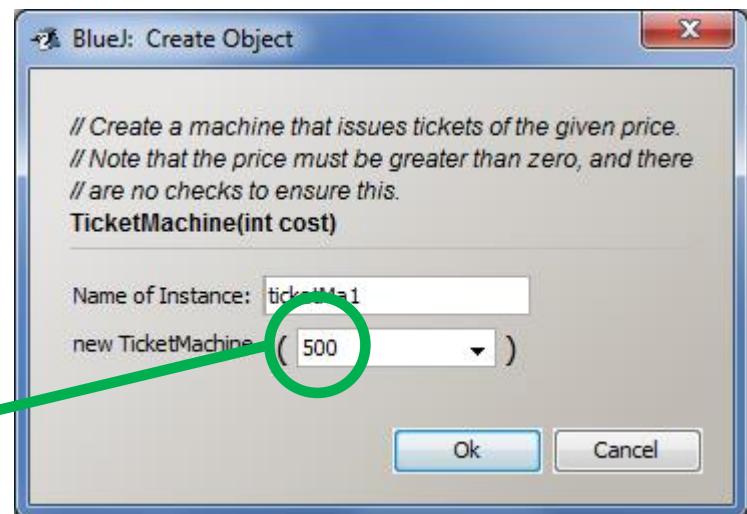
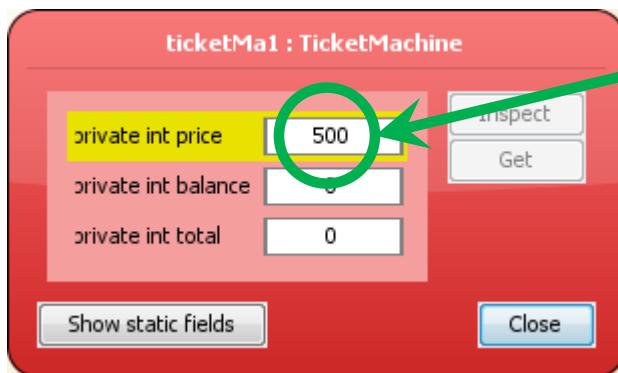
- Each constructor

**NOTE THE SEMI-COLONS!**

- Has the same name as the class
- Has no return type (contrast this with methods, later)
- Constructors *initialise* the object's fields
  - Sometimes using *defaults* (e.g. balance, total)
  - Sometimes from data passed in as *parameters* (e.g. price)
  - Sometimes there are multiple constructors which initialise the fields in different ways

# Passing data via parameters

```
public TicketMachine (int cost)
{
    price = cost;
    balance = 0;
    total = 0;
}
```

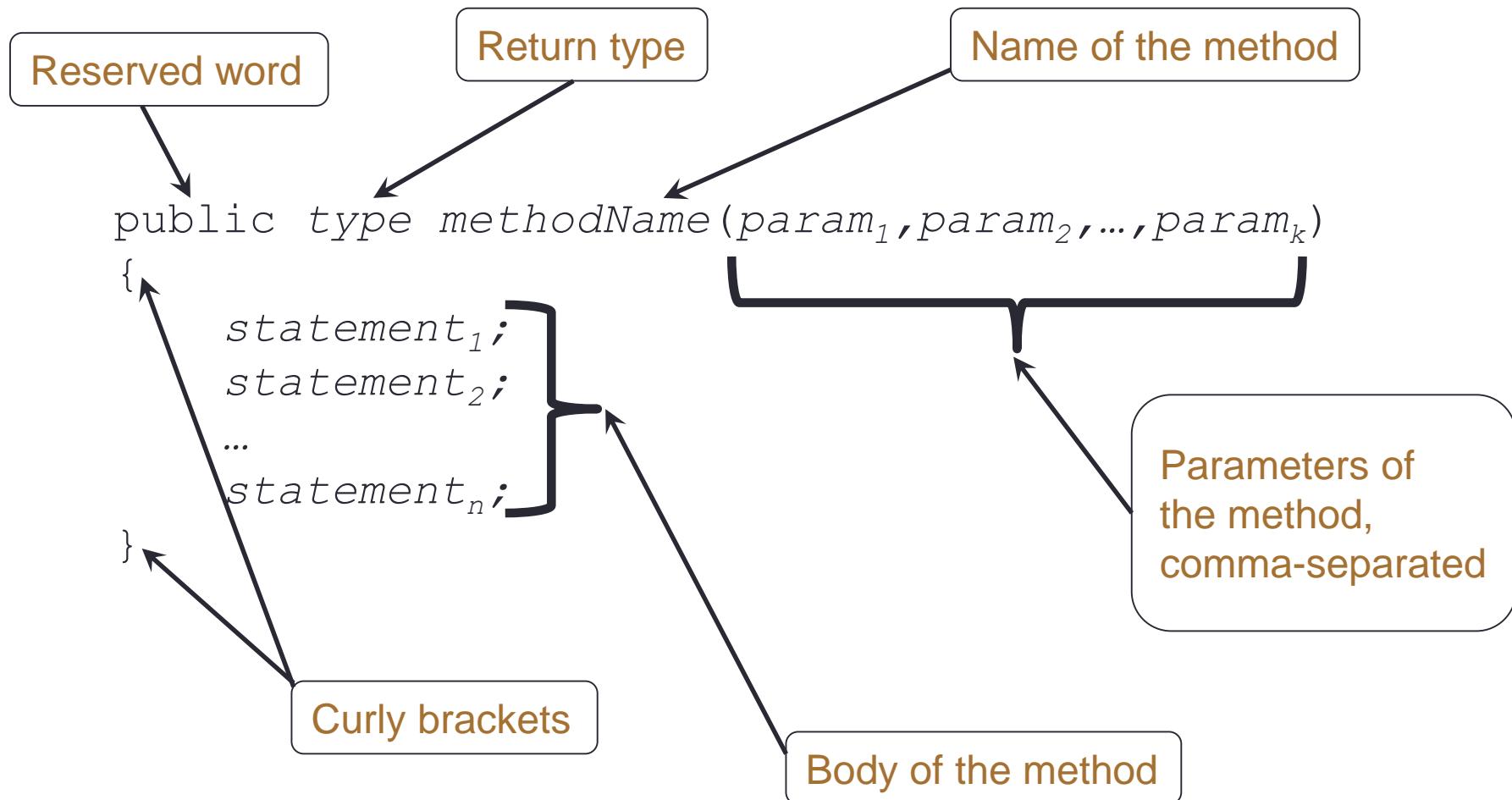


- Parameters are a second sort of variable (after fields)
  - We will meet other sorts of variables later in CITS1001

# Methods

- Methods implement the behaviour of objects
- Methods can implement any form of behaviour, as required by the class being implemented
  - The principal value of computers lies in their flexibility
- A method comprises a *header* and a *body*

# General method syntax

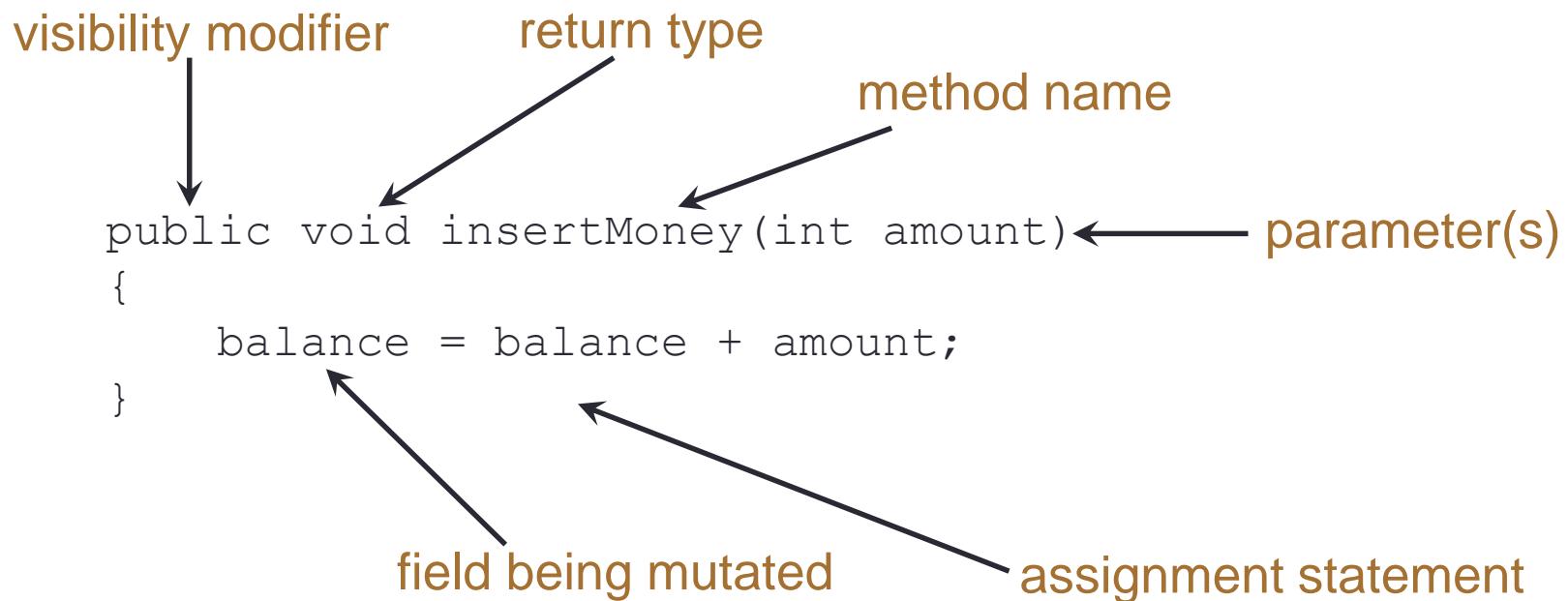


# Example methods

- We will introduce methods first via two relatively simple specific forms that are very common in Java programs
- *Mutator methods* alter the state of an object
- *Accessor methods* provide information about an object
- Remember that other sorts of methods can accomplish a very wide variety of tasks

# Mutator methods example

```
visibility modifier      return type      method name  
↓  
public void insertMoney(int amount) ← parameter(s)  
{  
    balance = balance + amount;  
}  
  
field being mutated      assignment statement
```



# Mutator methods

- They have the standard method structure
  - Header and body
- They are used to *mutate* (i.e. change) an object's state
  - Achieved through changing the value of one or more fields
- They usually have the return type `void`
- They typically contain assignment statements
- They often receive data through parameters
- Our example method takes a parameter representing money inserted to the ticket machine, and adds that to the current balance inside the machine
  - `balance` starts at 0 when an object is created, and it increases with each call to `insertMoney`

# Accessor (get) methods example

```
visibility modifier      return type      method name      empty parameter list  
public int getBalance ()  
{  
    return balance;  
}
```

visibility modifier → public

return type → int

method name → getBalance()

empty parameter list → ()

return statement → return balance;

# Accessor methods

- An accessor method returns a value (*result*) of the type given in the header
- Usually it just looks up the current value of one of the object's fields
  - Sometimes it does some minor calculation on that value
- An accessor method always has a return type that is not `void`
- The method contains a `return` statement to denote the value returned
  - NB: returning is *not* printing!
- Our example method allows you to query the current balance inside the machine

# set mutator methods

- Fields often have dedicated `set` mutator methods
- These have a simple, distinctive form
  - `void` return type
  - The method name is related to the field name
  - A single parameter, with the same type as the type of the field
  - A single assignment statement that sets the new value of the field

```
public void setPrice(int amount)
{
    price = amount;
}
```

# Protective mutators

- A set method does not have to simply assign the parameter to the field
  - The parameter may be checked for validity, and rejected if inappropriate
- Such mutators thereby protect fields

```
public void setPrice(int amount)
{
    if (amount > 0) {
        price = amount;
    }
}
```

- This version ensures that `price` is a positive number
  - If `amount` is positive, `price` is updated
  - If `amount` is zero or negative, nothing is changed

# Method summary

- Methods are used to implement all object behaviour
- A method has a name and a return-type
  - The return-type may be `void`
  - A non-`void` return type means the method returns a value to its caller
- A method might take parameters
  - Parameters bring values in from outside for the method to use

# CITS1001

## 4. CODING BASICS – PART 1

---

*Objects First With Java,*  
Chapter 2

# Lecture essentials

- Variables
- Assignment
- Types
- Type-checking
- Conditionals
- Expressions: arithmetic, relational, and logical
- Augmented assignment
- Working with Strings
- Variable scope

# Variables

- Variables are the basic mechanism by which data is organised and stored in all programming languages
  - Used for long-term storage, short-term storage, and communication
- Every variable in a Java program has a declaration, which determines
  - Its name, by which it is accessed and updated
  - Its type, which determines what values it can store and what operations it can participate in
  - Its scope, which determines where in the program it can be used; this is implicit from what kind of variable it is
- Field variables also have a visibility, which is an issue we will come to later in the unit

# Variables and assignment

- You can think of a variable declaration as creating a box in the store of the computer which holds a value

```
private int x;
```

- The value stored in the box can be updated using an assignment statement

```
x = 6;  
x = 2 * x;  
x = 2 * x;
```

- The RHS of the statement is evaluated, and the result is stored in the box for the variable on the LHS
- Note that assignment is *destructive*; the value in the box prior to the assignment is overwritten and is lost

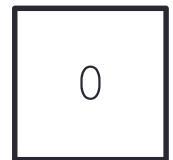
# Variables and assignment

- You can think of a variable declaration as creating a box in the store of the computer which holds a value

```
private int x;
```

- The value stored in the box can be updated using an assignment statement

```
x = 6;  
x = 2 * x;  
x = 2 * x;
```



- The RHS of the statement is evaluated, and the result is stored in the box for the variable on the LHS
- Note that assignment is *destructive*; the value in the box prior to the assignment is overwritten and is lost

# Variables and assignment

- You can think of a variable declaration as creating a box in the store of the computer which holds a value

```
private int x;
```

- The value stored in the box can be updated using an assignment statement

```
x = 6;  
x = 2 * x;  
x = 2 * x;
```



- The RHS of the statement is evaluated, and the result is stored in the box for the variable on the LHS
- Note that assignment is *destructive*; the value in the box prior to the assignment is overwritten and is lost

# Variables and assignment

- You can think of a variable declaration as creating a box in the store of the computer which holds a value

```
private int x;
```

- The value stored in the box can be updated using an assignment statement

```
x = 6;  
x = 2 * x;  
x = 2 * x;
```



- The RHS of the statement is evaluated, and the result is stored in the box for the variable on the LHS
- Note that assignment is *destructive*; the value in the box prior to the assignment is overwritten and is lost

# Variables and assignment

- You can think of a variable declaration as creating a box in the store of the computer which holds a value

```
private int x;
```

- The value stored in the box can be updated using an assignment statement

```
x = 6;  
x = 2 * x;  
x = 2 * x;
```



- The RHS of the statement is evaluated, and the result is stored in the box for the variable on the LHS
- Note that assignment is *destructive*; the value in the box prior to the assignment is overwritten and is lost

# Variable naming

- Variables should always be given meaningful names, for the benefit of people trying to read and understand your code
  - It is easier for readers to understand the meaning and intended use of `stepCount` than of `sss`
- Case is **ALWAYS** significant
  - `stepCount` and `stepcount` would be two different variables
- Variable names should always start with a lower-case letter
  - Makes it easy to distinguish them

# Variable naming errors

- If you use a variable name that BlueJ doesn't recognise, e.g.

```
int stepCount;  
stepcount = 99;
```
- You will get the error message  
*“Cannot find symbol – variable stepcount”*
- BlueJ is **ALWAYS** right!
  - You may have misspelled the name
  - You may have got the case wrong
  - You may have used the variable outside its scope
- But BlueJ is always right, no matter what you may think...

# Initialising a variable

- There is a shorthand for declaring and initializing a variable in one statement

```
int players; // declaration  
players = 15; // initialization
```

- This can be replaced by

```
int players = 15;
```

# Types

- Every variable in a Java program has a type, which tells you what range of values are allowed for the variable
- Java has *primitive* types and *object* types
- The four main primitive types are int, double, boolean, and char
  - It is also important to understand the operators available on each
- We will meet several built-in object types
  - The only one for now is String
- Defining classes also introduces new object types
  - To be discussed later

# Types: numbers

- The main type for integers is `int`, with a range of roughly  $\pm 2,000,000,000$ 
  - `byte`, `short`, and `long` denote integer variables with other ranges
- The main type for real numbers is `double`, with a range of roughly  $\pm 10^{308}$  and a precision of about 15 decimal digits
  - `float` denotes real variables with a smaller range and precision
- The only complication is with real numbers
  - The way that computers represent them limits their precision
  - In the BlueJ Code Pad, try the following

0.3

0.2

0.1

0.3 - 0.1

0.3 - 0.1 == 0.2

# Types: boolean

- boolean variables can take only two values, either true or false
  - Used to represent any binary situation
  - e.g. yes/no, correct/incorrect, accept/reject, etc.
  - Used mainly for recording decisions during program execution

```
int x = 2 * 6;  
boolean b1 = (x == 12);           // will set b1 to true  
boolean b2 = (x > 15);           // will set b2 to false  
int y = - x / 3;  
boolean b3 = (x + y <= 9);      // will set b3 to true
```

# Types: boolean

- boolean variables can take only two values, either true or false
  - Used to represent any binary situation
  - e.g. yes/no, correct/incorrect, accept/reject, etc.
  - Used mainly for recording decisions during program execution

**NOTE THE DIFFERENCE!**

```
int x = 2 * 6;
boolean b1 = (x == 12);           // will set b1 to true
boolean b2 = (x > 15);           // will set b2 to false
int y = - x / 3;
boolean b3 = (x + y <= 9);      // will set b3 to true
```

# Types: char

- `char` variables represent all types of characters
  - Letters (upper- and lower-case), digits, punctuation
  - Formatting characters, e.g. carriage-return '`\n`', tab '`\t`', etc.
  - Special characters, e.g. smiley, love heart, the Ace of Spades, etc.
  - Characters used in other languages, e.g. Arabic, Chinese, etc.
  - etc., etc., etc.
- In the (standard) Unicode system, every character is represented inside the machine by an integer
  - <http://www.tamasoft.co.jp/en/general-info/unicode-decimal.html>
- Characters are written in single quotes , e.g. '`A`', '`5`', '`;`'
  - They can also be written using their Unicode values in hexadecimal, e.g. '`\u0041`' is the same character as '`A`'
- Much more about characters later in the unit

# Types: String

- String that we have seen already is an object type which represents a sequence of characters
  - Used to represent names, addresses, general text, etc.
- Strings are written in double quotes
  - e.g. “CITS1001” is a sequence of length 8
  - e.g. “A” is a sequence of length 1
  - “” is a sequence of length 0
  - But note that “ ” is a sequence of length 1 containing a space
  - And “ ” is a sequence of length 2 containing two spaces...

# Syntax is important!

- Be clear on the difference between e.g.
  - 8, which is an int
  - '8', which is a char
  - "8", which is a String of length 1
- And between e.g.
  - 184, which is an int
  - "184", which is a String of length 3
- And between e.g.
  - 1.84, which is a double
  - "1.84", which is a String of length 4
- And between e.g.
  - "noOfUnits", which is a String of length 9
  - noOfUnits, which is valid only if there is a variable with this name

# Type-checking

- The principal use of types in Java is so the computer can help you to find some kinds of errors in your program
- When you compile your program, one thing that BlueJ does is *type-checking*
  - It checks that every expression has argument(s) of the right type(s)
  - It checks that every assignment has the right type
  - It checks that every argument passed into a method or constructor has the right type

# Type-checking

- All of the following will cause errors during compilation
  - “Incompatible types”

```
int x;  
boolean b;  
x = true;           // can't assign a boolean to x  
x = 4 + 2.3;       // can't assign a double to x  
x = "007";         // can't assign a String to x  
b = 55 - 33;       // can't assign an int to b  
insertMoney("87"); // insertMoney expects an int  
insertMoney(b);    // ditto  
insertMoney(4 + true); // can't apply + to a boolean
```

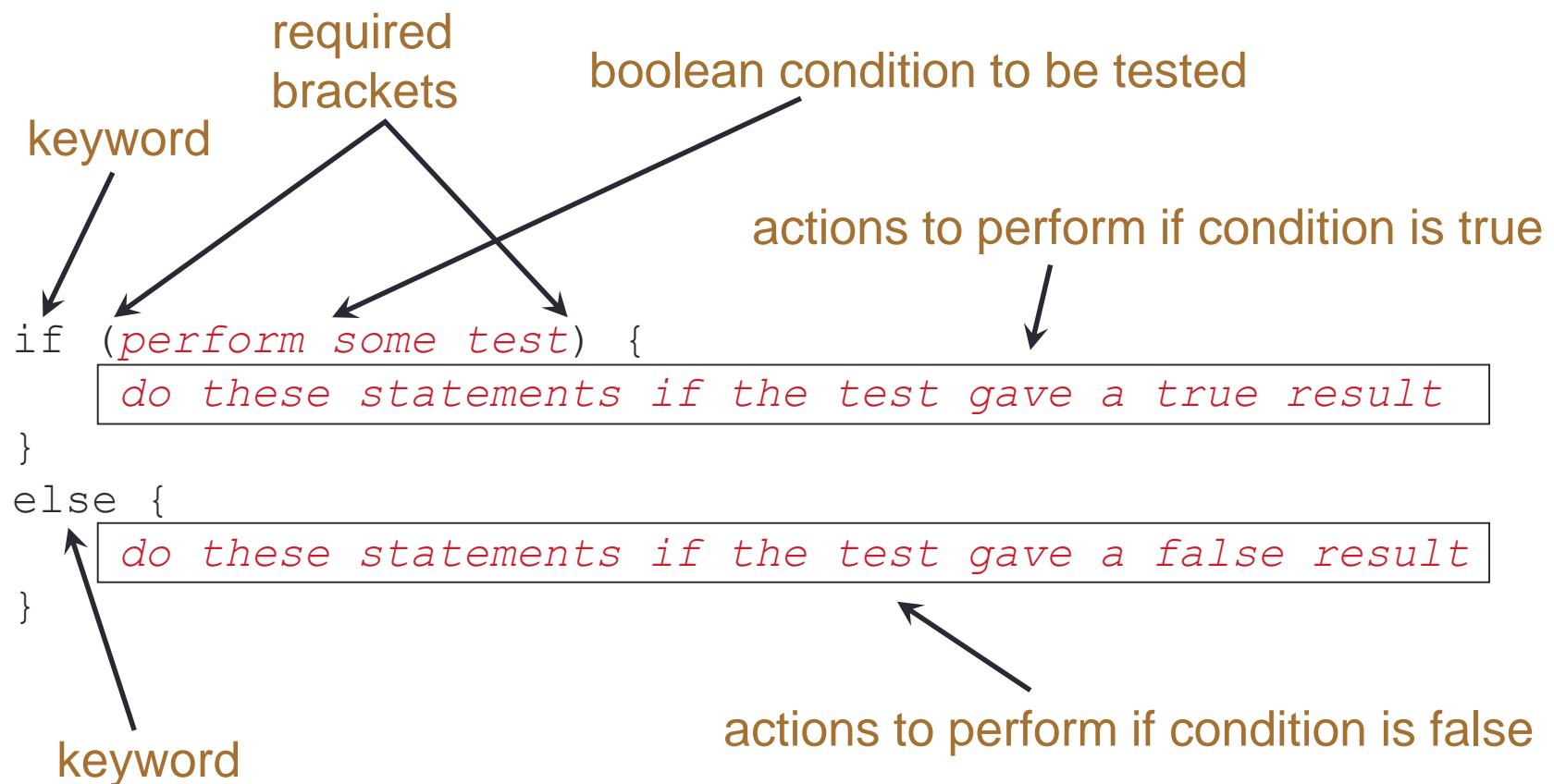
# Conditional behaviour

- Sometimes life involves choices
- “If I have enough money left on the weekend, I will go out to eat, otherwise I will stay in and watch a movie.”

```
if (I have enough money left) {  
    // go out for a meal  
}  
else {  
    // stay home and watch a movie  
}
```

- Note that the result depends on the amount of money I have *at the time of the decision*

# Making choices in Java



# Execution

- When the conditional statement is encountered, the boolean expression is evaluated
  - It returns either true or false
- If the result is true, the first statement block is executed, and the second is skipped
- If the result is false, the first statement block is skipped, and the second is executed
- In every case
  - Exactly one of the blocks is executed
  - Execution continues with the statements following the conditional (if any)

# Flow of execution

```
<statements1>
if (condition) {
    <statements2a>
}
else {
    <statements2b>
}
<statements3>
```

- The order of execution here is
  - <statements1>, **then**
  - <condition>, **then**
  - **either** <statements2a> **or** <statements2b>, **then**
  - <statements3>

# Conditional statement example

- The original version of `insertMoney` in *TicketMachine* permits negative arguments
  - Essentially withdrawals!

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

- If `amount` is negative, `balance` will get smaller
  - It could even go negative itself!

# Conditional statement example

- Better to reject negative arguments and print an error message
  - Then balance can only increase

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

# Sometimes there's only one optional action

- “If I have enough money next year, I will buy a new bike.”
  - No explicit alternative action if money is tight
- `discountPrice` means to reduce the ticket price by `diff`

```
public void discountPrice(int diff)
{
    price = price - diff;
}
```

# Contd.

- But we don't want to accidentally increase the price

```
public void discountPrice(int diff)
{
    if (diff > 0) {
        price = price - diff;
    }
}
```

- Now the price will only ever be reduced
  - We don't need an `else` clause really
- If the condition evaluates to `true`, the protected statement block is executed
  - Otherwise it is skipped

# Contd.

- Also we don't want the price to go to zero or beyond!

```
public void discountPrice(int diff)
{
    if (diff > 0 && diff < price) {
        price = price - diff;
    }
}
```

- Or equivalently

```
public void discountPrice(int diff)
{
    if (diff > 0) {
        if (diff < price) {
            price = price - diff;
        }
    }
}
```

## Sometimes there are more than two optional actions

- At UWA, a mark over 80 gets you a HD, 70–79 gets you a D, 60–69 gets you a CR, 50–59 gets you a P, less gets you an F

```
public String grade(int mark)
{
    String g;
    if (mark >= 80) {g = "HD";}
    else
        if (mark >= 70) {g = "D";}
        else
            if (mark >= 60) {g = "CR";}
            else
                if (mark >= 50) {g = "P";}
                else {g = "F";}
    return g;
}
```

# In general

```
if (perform test1) {  
    do these statements if test1 gave a true result  
}  
else  
if (perform test2) {  
    do these statements if test1 false and test2 true  
}  
else  
if (perform test3) {  
    do these statements if tests 1-2 false and test3 true  
}  
else  
if (perform test4) {  
    do these statements if tests 1-3 false and test4 true  
}  
else {  
    do these statements if all four tests gave a false result  
}
```

# An incorrect version

- This “version” of the UWA grading method is wrong
  - It is missing all the `elses`
- Make sure you understand how it goes wrong
  - And what it returns for various values of `mark`

```
public String gradeWRONG(int mark)
{
    String g;
    if (mark >= 80) {g = "HD";}
    if (mark >= 70) {g = "D";}
    if (mark >= 60) {g = "CR";}
    if (mark >= 50) {g = "P";}
                           {g = "F";}
    return g;
}
```

To be continued in Part 2

# CITS1001

## 4. CODING BASICS – PART 2

---

*Objects First With Java,*  
Chapter 2

# Lecture essentials

- Variables
- Assignment
- Types
- Type-checking
- Conditionals
- Expressions: arithmetic, relational, and logical
- Augmented assignment
- Working with Strings
- Variable scope

# Expressions

- Expressions are used in programming in largely the same way as in mathematics
  - To create values by combining other values
- We have already seen many simple examples
  - e.g. on the previous few slides
- Every expression has a corresponding *value* and a *type*
- We introduce here four kinds of expressions in Java
  - Literals
  - Names
  - Method invocations
  - Compound expressions

# Literals

- Literals are values “hard-wired” into the language
  - e.g. 2, -2, 3.14159, 10e-5, true, '.', '\n'
- The value and type of the expression are just the value and type of the literal
  - i.e. int, int, double, double, boolean, char, char

# Names

- Names are the declared *variables* that are in scope
  - e.g. (from our earlier examples) `stepCount`, `amount`, `players`, `x`
- The value and type of the expression are the type of the variable and its current value
  - All of the above are `int` in our examples from earlier

# Method invocations

- A call to a non-void method of some object
  - The type of the expression is the return type of the method
  - The value of the expression is the returned value
- e.g. from *TicketMachine*, `getBalance()`
  - Its type is `int`, because the return type of the method is `int`
  - Its value is the current value of the object's balance
- We also may have to identify which object is being queried
  - More on this later

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

# Compound expressions

- A compound expression has an *operator* applied to one or more smaller expressions, known as *arguments*
  - e.g. `3+5`
  - The type of this is `int`, and its value is `8`
- The principal operators fall into three groups
  - Arithmetic operators combine numbers
  - Relational operators test values and return booleans
  - Logical operators combine booleans
- All binary operators have *precedence* and *associativity*
  - Given `x op1 y op2 z`, precedence tells you whether this is  $(x \text{ op1 } y) \text{ op2 } z$  or  $x \text{ op1 } (y \text{ op2 } z)$
  - Given `x op1 y op1 z`, associativity tells you whether this is  $(x \text{ op1 } y) \text{ op1 } z$  or  $x \text{ op1 } (y \text{ op1 } z)$

# Arithmetic operators

- The principal arithmetic operators are `+`, `-`, `*`, `/`, and `%`
  - `E1 % E2` returns the remainder left after `E1` is divided by `E2`
  - e.g. `8 % 5` evaluates to `3`
  - e.g. `8 % 2.5` evaluates to `0.5`
- Precedence and associativity are as normal in Maths
- The rule for types is that
  - If both arguments are `int`, the result is an `int`
  - If either or both arguments are `double`, the result is a `double`
- e.g. `8+5` returns the `int` `13`
- e.g. `8+5.0`, `8.0+5`, and `8.0+5.0` all return the `double` `13.0`

# Arithmetic operators confusion

- The principal source of confusion tends to be when `int` and `double` are combined with division
- Division on `int` truncates
  - e.g. `8/5` returns the `int` `1`, as do both `501/501` and `1001/501`
- Division on `double` is exact
  - e.g. `8/5.0`, `8.0/5`, and `8.0/5.0` all return the `double` `1.6`
- What about `2.0 * 4 / 5?`
- What about `4 / 5 * 2.0?`
- What about `2.0 + 4 / 5?`
- What value does `x` have after this statement?  
`double x = 24 / 10;`

# Relational operators

- These are used to compare two values
  - Usually numbers, but also sometimes other types
- There are six principal relational operators

E1 == E2

Is E1 equal to E2?

E1 != E2

Is E1 not equal to E2?

E1 < E2

Is E1 less than E2?

E1 <= E2

Is E1 less than or equal to E2?

E1 > E2

Is E1 greater than E2?

E1 >= E2

Is E1 greater than or equal to E2?

- They all have lower precedence than all arithmetic operators, and higher than all logical operators
- e.g. 10 - 5 < 4 \* 1.23 evaluates to false

# Logical operators

- There are three logical operators

`E1 && E2`      Are E1 and E2 both true?      (*and*)

`E1 || E2`      Is at least one of E1 and E2 true?      (*or*)

`!E`      Is E false?      (*not*)

- `&&` has a higher precedence than `||`
- e.g. `10 < 15 - 5 || 4 * 0.75 >= 3 && !(8 > 3)`  
evaluates to `false`
  - Check it out!

# Augmented assignment

- Java supports augmented assignment for common arithmetic and logical operators, e.g.

<code>x += E;</code>	is equivalent to	<code>x = x + E;</code>
<code>x -= E;</code>	is equivalent to	<code>x = x - E;</code>
<code>x *= E;</code>	is equivalent to	<code>x = x * E;</code>
<code>x /= E;</code>	is equivalent to	<code>x = x / E;</code>
<code>x %= E;</code>	is equivalent to	<code>x = x % E;</code>
<code>x &amp;= E;</code>	is equivalent to	<code>x = x &amp;&amp; E;</code>
<code>x  = E;</code>	is equivalent to	<code>x = x    E;</code>

# Operators over String

- Two important operations over strings are
  - Concatenation – joining strings together
  - Printing – displaying strings on the screen
- These two operations are frequently used in combination
- Java provides many, many other operations over strings
  - We will see these later in the unit

# String concatenation

- In Java, concatenation is achieved with the + operator

```
int p = 500; int cents = 25; String s = "";
```

5 + 6 → 11

"wind" + "ow" → "window"

55 + "parts" → "55parts"

"Week " + 55 + s → "Week 55"

"#" + p + "cents" → "#500cents"

5 + "hello" + 6 → ?

"hello" + 5 + 6 → ?

5 + 6 + "hello" → ?

# String printing

- In Java, printing is performed mainly with two operators
  - `System.out.print(s)` displays `s` in the terminal window
  - `System.out.println(s)` displays `s` in the terminal window and adds a carriage-return on the end

`System.out.println(s)`

is equivalent to

`System.out.print(s + "\n")`

- It is usually clearest if different lines of output are sent to the screen with different calls to `println`
  - e.g. `printTicket`

# String printing

- In Java, printing is performed by:
  - `System.out.print(s)`
  - `System.out.println(s)` which prints `s` and adds a carriage-return

`System.out.println(s)`

`System.out.print(s +`

- It is usually clearest if different parts of a message go to the screen with different statements
  - e.g. `printTicket`

```
/**  
 * Print a ticket.  
 * Update the total collected and  
 * reduce the 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;  
}
```

# Kinds of variables and their scope

- There are three basic kinds of variables, all illustrated by this simple code fragment

```
public class Example1
{
    private int total;
    // constructor omitted
    public void deposit(int a)
    {
        int diff;
        diff = total - a;
        total = total + diff;
    }
}
```

- **total is a field variable, part of an object's state**
  - Long-term storage
- **Its scope is the entire class**
  - i.e. it can be accessed anywhere in the class

# Kinds of variables and their scope

- There are three basic kinds of variables, all illustrated by this simple code fragment

```
public class Example1
{
    private int total;
    // constructor omitted
    public void deposit(int a)
    {
        int diff;
        diff = total - a;
        total = total + diff;
    }
}
```

- a **is a parameter**
  - Storage for communication
- **Its scope is the method it is passed into**
  - i.e. it can be accessed only inside the method `deposit`

# Kinds of variables and their scope

- There are three basic kinds of variables, all illustrated by this simple code fragment

```
public class Example1
{
    private int total;
    // constructor omitted
    public void deposit(int a)
    {
        int diff;
        diff = total - a;
        total = total + diff;
    }
}
```

- **diff is a local variable**
  - Short-term storage
- **Its scope is the method it is declared in**
  - i.e. it can be accessed only inside the method `deposit`

# Different variables can have the same name

- Here there are two *different* variables called `a`

```
public class Example2
{
    // constructor elided
    public int m1(int a)
    {
        return a + 4;
    }

    public int m2(int a)
    {
        return a - 4;
    }
}
```

- This is fine, because the scope of each variable is separate
  - Each can be accessed only inside its own method
  - It is clear which variable each use refers to

# “Variable shadowing”

- Here there are two *different* variables called `x`, but their scopes overlap

```
public class Example3
{
    private int x;

    // constructor elided

    public void m1(int x)
    {
        // try to set the field x
        x = x;
    }
}
```

- This also is legal Java, but potentially confusing
  - Inside `m1`, both can be accessed
- Java considers that both uses of `x` in the assignment refer to the parameter
- The field is unchanged by this code
  - In fact the assignment does nothing
- This is easy to do accidentally

# Two alternative solutions

- Use different names for the variables, or use this

```
public class Example3
{
    private int x;

    // constructor elided

    public void m1(int x1)
    {
        // set the field x
        x = x1;
    }
}
```

```
public class Example3
{
    private int x;

    // constructor elided

    public void m1(int x)
    {
        // set the field x
        this.x = x;
    }
}
```

**this.var always refers to a field variable**

# CITS1001

## 5. OBJECT INTERACTION

---

*Objects First With Java,*  
Chapter 3

# Lecture essentials

- Abstraction and modularization in software design
- Classes define types
- Objects can create objects
- Internal/external method calls
- null references

# A digital clock

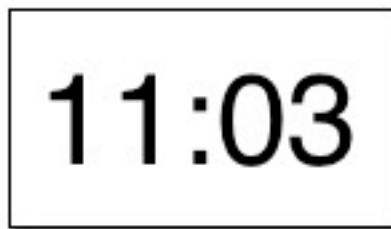
- A case study from *Objects First*



# Abstraction and modularization

- *Abstraction* means ignoring the details or the parts of a problem, to focus attention on its higher levels
  - cf. User-view vs. writer-view of a class
  - The writer must worry about details; the user cares only how to use the class and what it does
- *Modularization* or *decomposition* means dividing a whole into well-defined parts, which
  - Can be built and examined separately
  - Interact in well-defined ways
  - Can be treated as “units” at higher levels of the design
  - cf. car design

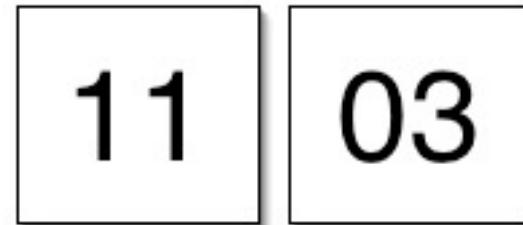
# Modularizing the clock display



One 4-digit display?

- We would need to solve all of the clock issues in one context

Or two 2-digit displays?



- How would the behaviour of these two displays compare?

# Fields in NumberDisplay

- Complete listings of both classes are on the LMS

```
public class NumberDisplay
{
    // value at which the counter resets
    private int limit;
    // current value of the counter
    private int value;
```

*Constructor and methods to come later.*

```
}
```



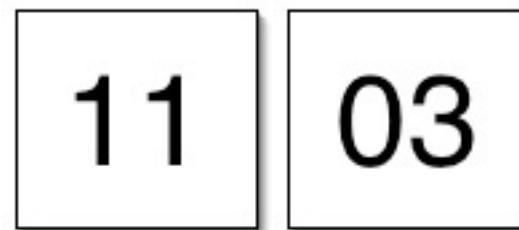
# Fields in ClockDisplay

```
public class ClockDisplay
{
    // the counter for hours
    private NumberDisplay hours;
    // the counter for minutes
    private NumberDisplay minutes;
    // used to record the current display
    private String displayString;
```

Two separate  
number displays

*Constructor and methods to come later.*

}



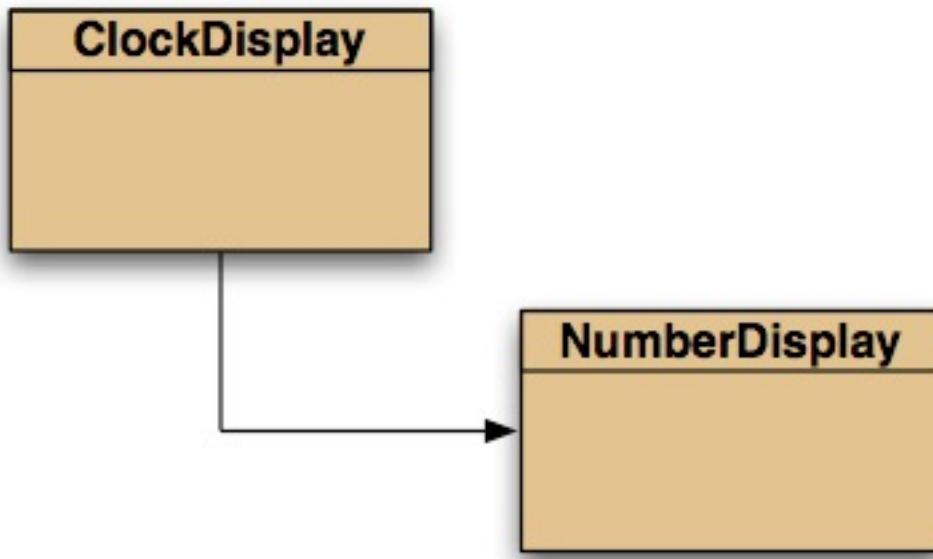
# Classes define types

- A class name can be used as the type for a variable

```
private NumberDisplay hours;
```

- Variables that have a class as their type can store references to objects belonging to that class
- This applies to both
  - User-defined classes like `NumberDisplay` in this example
  - Built-in library classes like `java.util.Random` in the *Guess the Number* case study
- More on library classes in Week 6

# Class diagram

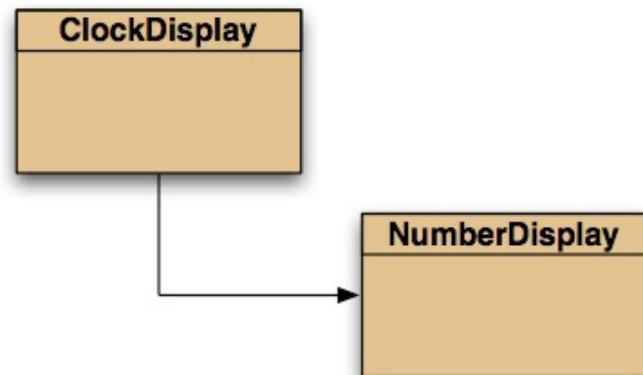


The class **ClockDisplay** *depends on* the class **NumberDisplay**

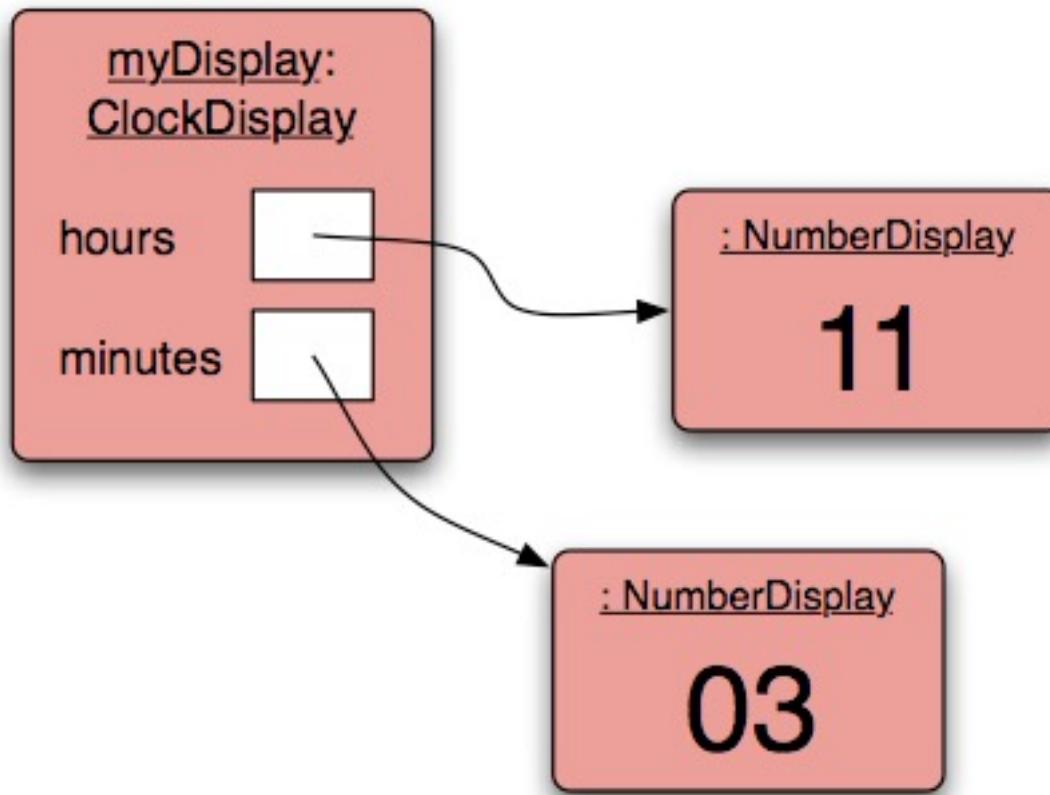
The class **ClockDisplay** *makes use of* the class **NumberDisplay**

# Class diagram

- Classes exist at compile-time
- The *class diagram* shows
  - The classes of an application, and
  - The relationships between them
- It gives information about the source code
- It presents a *static* view of the program



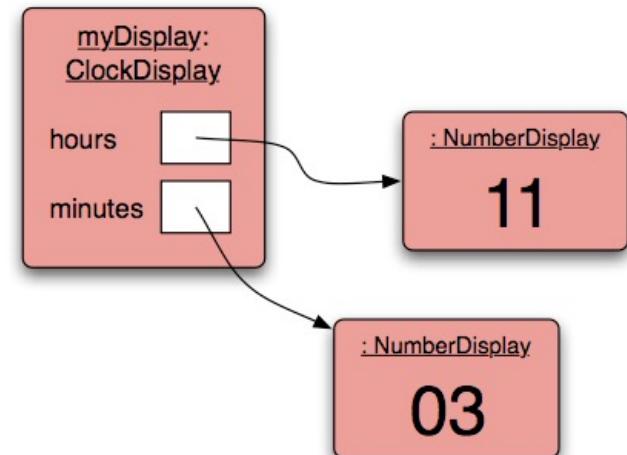
# Object diagram



The object `myDisplay` can refer to its `NumberDisplay` object fields via the variables `hours` and `minutes`

# Object diagram

- Objects exist at run-time
- The *object diagram* shows
  - The objects existing at one moment during an execution of the program, and
  - The relationships between them
- It gives information about the objects at run-time
- It presents a *dynamic* view of the program



# Source code for NumberDisplay

```
public class NumberDisplay
{
    // value at which the counter resets
    private int limit;
    // current value
    private int value;

    public NumberDisplay(int limit)
    {
        this.limit = limit;
        value = 0;
    }

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

Think about what increment does as value approaches limit

# Source code for ClockDisplay

```
public class ClockDisplay
{
    // current number of hours
    private NumberDisplay hours;
    // current number of minutes
    private NumberDisplay minutes;
    // used to record the current display
    private String displayString;

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

    Methods to come later.
}
```

# Objects creating objects

- Objects are created by applying `new` to a constructor
- In class `ClockDisplay`

```
hours = new NumberDisplay(24);
```

- 24 is an *actual parameter* or *argument*
- In class `NumberDisplay`

```
public NumberDisplay(int limit) {...}
```

- limit is a *formal parameter*, used to initialise the field

# Objects creating objects

- Make sure you understand the relationship between these two lines in `ClockDisplay`

```
private NumberDisplay hours;  
...  
hours = new NumberDisplay(24);
```

- The first line declares the variable `hours`
- In the second line,
  - The right-hand side creates a `NumberDisplay` object
  - The assignment makes `hours` points to that object
- **If you haven't called `new`, you haven't created an object**

# Objects creating objects

- Make sure you understand the relationship between these two lines in ClockDisplay

```
private NumberDisplay hours;  
...  
hours = new NumberDisplay(24);
```



- The first line declares the variable hours
- In the second line,
  - The right-hand side creates a NumberDisplay object
  - The assignment makes hours points to that object
- **If you haven't called new, you haven't created an object**

# Objects creating objects

- Make sure you understand the relationship between these two lines in ClockDisplay

```
private NumberDisplay hours;  
...  
hours = new NumberDisplay(24);
```

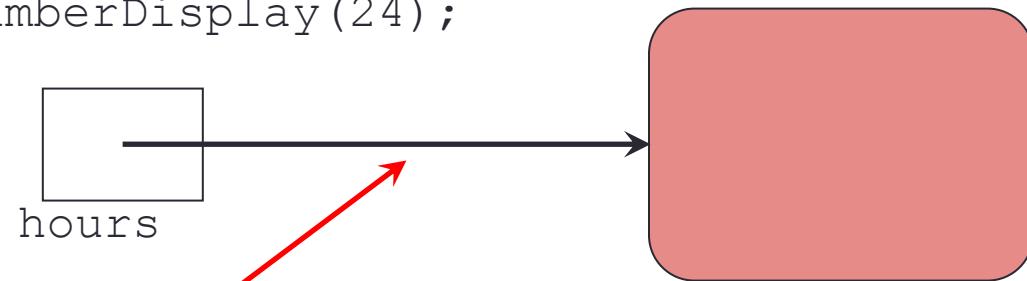


- The first line declares the variable hours
- In the second line,
  - The right-hand side creates a NumberDisplay object
  - The assignment makes hours points to that object
- If you haven't called new, you haven't created an object

# Objects creating objects

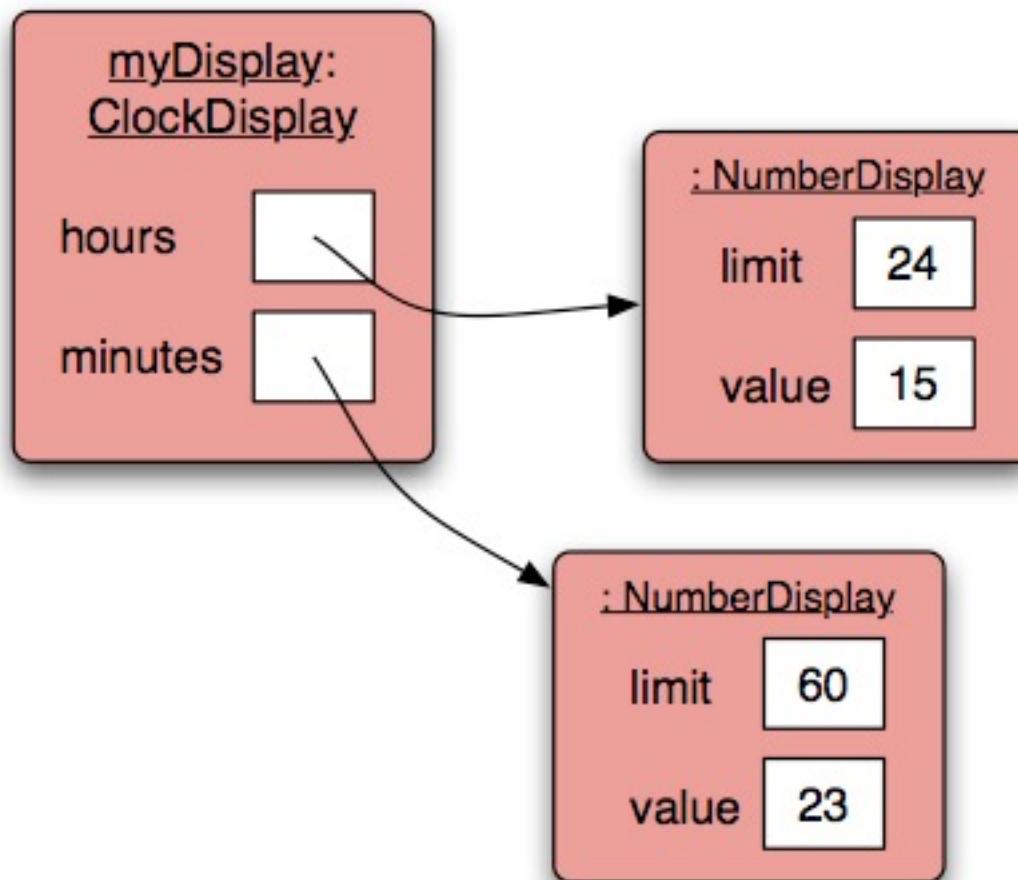
- Make sure you understand the relationship between these two lines in ClockDisplay

```
private NumberDisplay hours;  
...  
hours = new NumberDisplay(24);
```



- The first line declares the ~~variable~~ hours
- In the second line,
  - The right-hand side creates a NumberDisplay object
  - The assignment makes hours point to that object
- **If you haven't called `new`, you haven't created an object**

# Object diagram



# Method calling in ClockDisplay

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

private void updateDisplay()
{
    displayString = hours.getDisplayValue() + ":" +
                    minutes.getDisplayValue();
}
```

# External method calls

- The calls to `increment`, `getDisplayValue`, and `getValue` are to the methods belonging to the `NumberDisplay` objects
  - Each call must specify which object it refers to
  - cf. if I ask “what’s your student number?”, I need to identify which student I am asking
- The general syntax for an external method call is

*objectName.methodName(parameter-list)*

The name of the object

The name of the method

Arguments required

# Internal method calls

- The call to `updateDisplay` is a call to the method belonging to the `ClockDisplay` object
  - Because it is to the same object, the method name alone is sufficient
  - cf. if I tell myself to remember my laptop, I know it's me!
- It is ok to include the object name `this` if you want

`this.methodName(parameter-list)`

- But usually it serves no purpose

# Method calls – mutating with increment

- The ClockDisplay method timeTick calls increment for each of its NumberDisplay objects
  - This is a mutator method that causes the object to change its value field
  - As a mutator method, it returns nothing to the caller

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

# Method calls – returning and receiving results

- `getDisplayValue` **returns the appropriate String from each of the NumberDisplay objects**

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

- In `ClockDisplay`, `updateDisplay` concatenates these values to make the complete display

```
private void updateDisplay()  
{  
    displayString = hours.getDisplayValue() + ":" +  
                  minutes.getDisplayValue();  
}
```

# null object references

- Object variables don't always refer to actual objects
- We saw earlier that when an object variable is first declared, it is initialised to the special value `null`
  - In the same way as `int` variables are initialised to `0`
- `null` means “this variable doesn't point to an object”
- You will sometimes see run-time error messages like `java.lang.NullPointerException`
- This message means you have tried to use an object variable when its value is `null`, e.g.

```
private NumberDisplay seconds;  
seconds.increment();
```

ERROR! `seconds` is null

# null error messages

- Object variables can be tested for being null

```
if (seconds != null) seconds.increment();
```

- No error!
- It may also be appropriate sometimes to explicitly set an object variable to be null

```
seconds = null;
```

- e.g. to “forget” a previous object

# Anonymous objects

- Sometimes objects are created and then immediately “passed on” as arguments
  - e.g. if lots.add takes an object belonging to class Lot as a parameter, we might write something like

```
Lot furtherLot = new Lot(...);  
lots.add(furtherLot);
```

- We are using the variable furtherLot **only** to pass the argument to add
- It is neater to omit the variable

```
lots.add(new Lot(...));
```

# Another expression-type

- Technically, new Constructor (...) is an expression like any other
- The previous example is logically equivalent to replacing

```
int x = y + 1;  
insertMoney(x);
```

with

```
insertMoney(y + 1);
```

# An exercise

- Extend the clock display so that it also records the day of the week
- Use an encoding where each day is represented by an integer (e.g. Monday = 0, Tuesday = 1, etc.)
  - So 9am Wednesday will look like “02 – 09:00”
- In ClockDisplay, you will need
  - An extra field to record the current day
  - Updated versions of the constructor, timeTick, and updateDisplay
    - What limit is appropriate for days of the week?
- What changes are needed in NumberDisplay?

# CITS1001

## 6. GUESS THE NUMBER – A SMALL CASE STUDY

---

# Lecture essentials

- A small case study on how to design a class to play a simple game
  - Uses many of the ideas described in the first three weeks of the unit
- Code is available on the LMS
  - An extension is suggested on the last slide as a small exercise for you
- Illustrates that there are often many reasonable ways to write a piece of code!
  - Also introduces the use of a *library class*; more on these in later lectures

# A case study – *guess the number*

- In a classic children's game, Bob chooses a number in the range 1–100, and June repeatedly tries to guess it
- Each of June's guesses gets a response from Bob
  - Either *too high*, or *too low*, or *well done!*
  - June tries to guess the number in as few goes as she can
- How would we write a Java class to play this game?
  - The computer will be Bob
- Each object belonging to the class will play one game, so it needs to choose a number randomly and then respond to each of the player's guesses
- We need to specify the class's instance variable(s), its constructor(s), and its method(s)
- Think about the problem before you read on...

# Instance variable

- We need only one instance variable, to store the chosen number

```
public class GuessTheNumber {  
  
    // holds the chosen secret number  
    private int secretNumber;  
  
}
```

# Constructor

- The constructor needs to choose the number
  - i.e. it needs to initialise the instance variable
- But how do we choose a random number?
  - Using a library class

```
public GuessTheNumber () {  
    java.util.Random r;  
    r = new java.util.Random();  
    secretNumber = r.nextInt(100) + 1;  
}
```

Declares a local variable `r` to hold a random number generator

Creates a new RNG and assigns it to `r`

Asks `r` for a random `int` between 0 and 99, then adds 1 to get a number between 1 and 100

# Library classes

- We can generate random numbers using a library class from the Java API (Application Programming Interface)
- In BlueJ, under the *Help* menu, select *Java Class Libraries*
- This opens <https://docs.oracle.com/en/java/javase/11/docs/api/> (or similar URL; older versions of BlueJ may vary)
- Click on *All Classes* in the top-left to see a list of the library classes available
- There are a lot! In CITS1001 you will likely use only 10–20 of these
- The important skill is to able to find information quickly when needed

# Random numbers in the Java API

- Scroll down to the class Random and click on it
- There's the name of the class

## Class Random

java.lang.Object  
java.util.Random

- It is called `java.util.Random` because the class Random is in a *package* of classes called `java.util`

# Random numbers in the Java API

- Scroll down to *Constructor Summary*
- There's the constructor we need

*Constructor Summary*

**Constructors**

**Constructor and Description**

**Random()**  
Creates a new random number generator.

- Very simple – no arguments required!

# Random numbers in the Java API

- Scroll down to *Method Summary*
- There's a method that will choose a random `int` for us

```
int          nextInt(int bound)
Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive),
drawn from this random number generator's sequence.
```

- Returns an integer between 0 and `bound-1`
  - So `nextInt(100)` returns an integer in the range 0..99
- If you click on the name of the method, you can also see the code, but don't worry about this
- Other methods in `Random` would also do the job

# Abstraction

- Notice the principle of *abstraction* at work here
  - We do not need to know *how* `java.util.Random` works
- We are simply users of this class, so we need to know
  - How to declare a variable (i.e. what is the class called?)
  - How to create an object (i.e. what are the constructor's arguments?)
  - How to ask it for a number (i.e. what's an appropriate method?)
- In general, the quickest way to solve a problem is if someone else has already written the code for you!
  - You need to learn to find and use these classes to become productive
  - The libraries are your friends!

# Method

- We need only one method, to respond to a guess

```
public String checkMyGuess (int guess) {  
    String response;←  
    if (guess > secretNumber) {  
        response = "too high";  
    }  
    else  
        if (guess < secretNumber) {  
            response = "too low";  
        }  
    else {  
        response = "well done!";  
    }  
    return response;  
}
```

The method's parameter

A local variable in the method

The object's instance variable

# Or...

- There are always many ways to write a piece of code
- Pre-initialise the return variable

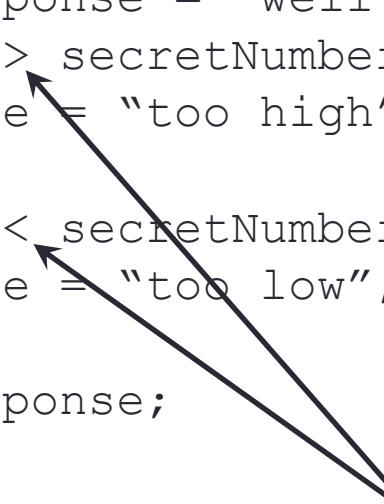
```
public String checkMyGuess (int guess) {  
    String response = "well done!";  
    if (guess > secretNumber) {  
        response = "too high";  
    }  
    else  
        if (guess < secretNumber) {  
            response = "too low";  
        }  
    return response;  
}
```

response gets updated if a different message is required

# Or...

- Pre-initialise, and don't nest the conditionals

```
public String checkMyGuess (int guess) {  
    String response = "well done!";  
    if (guess > secretNumber) {  
        response = "too high";  
    }  
    if (guess < secretNumber) {  
        response = "too low";  
    }  
    return response;  
}
```

Two black arrows originate from the 'secretNumber' variable in the two nested if statements and point towards a yellow callout box at the bottom right.

Clearly at most one of the conditions will be true

# Or...

- Pre-initialise, don't nest, and save the { }

```
public String checkMyGuess (int guess) {  
    String response = "well done!";  
    if (guess > secretNumber)  
        response = "too high";  
    if (guess < secretNumber)  
        response = "too low";  
    return response;  
}
```

If one of the alternatives in a conditional contains only one statement, the { } are not strictly necessary

# Or...

- Don't use a return variable, with nesting

```
public String checkMyGuess (int guess) {  
    if (guess > secretNumber)  
        return "too high";  
    else if (guess < secretNumber)  
        return "too low";  
    else return "well done!";  
}
```

The method terminates as soon as  
it executes any `return` statement

# Or...

- No return variable, and no nesting

```
public String checkMyGuess (int guess) {  
    if (guess > secretNumber)  
        return "too high";  
    if (guess < secretNumber)  
        return "too low";  
    return "well done!";  
}
```

# But NOT

- No return variable, no nesting, and all three checks

```
public String checkMyGuess (int guess) {  
    if (guess > secretNumber)  
        return "too high";  
    if (guess < secretNumber)  
        return "too low";  
    if (guess == secretNumber)  
        return "well done!";  
}
```

Java is being too careful – it can't  
guarantee a `return` will be executed

# An enhancement

- Make sure you understand how all of these versions always return the same result
  - Except the last one, of course
- Suppose we want the class to count (and report) the number of guesses taken to find the number
  - What do we need to do?
- Again, think about the problem before you read on

# An enhancement

- We need a field to record the number of guesses so far
- The constructor needs to initialise the field
- The method needs to update the field
- The method needs to print a message when the game is over
- Try this for homework

# CITS1001

## 7. GROUPING OBJECTS

---

*Objects First With Java,  
Chapter 4*

# Lecture essentials

- Data often exists in collections
- Java provides the library class `ArrayList` to manage collections of objects
- `ArrayList` is a very flexible class providing a wide range of operations for users
- For-each loops allow users to process all or some of the items in a collection

# The need to group objects

- Many applications involve maintaining collections of information
  - Personal organizers and smartphones
  - Library and store catalogues
  - Student-record systems and other databases
  - Games and artificial worlds
  - Web browsers and servers
  - *etc., etc., etc.*
- Frankly almost all interesting applications!

# Functionality required

- Typically we need the ability to
  - Create a collection
  - Add items to the collection
  - Remove items from the collection
  - Query the collection, e.g. for its current size
  - Access items individually
  - Process all of the items
  - Process selected items
- Ideally also there will be
  - No need to specify the size of a collection in advance
  - No limit on the maximum size of a collection

# ArrayList

- All of this functionality is provided by the Java library class `java.util.ArrayList`
- This is a *parameterized class*
  - An `ArrayList` can hold objects of any type
  - But all objects in one `ArrayList` must be the same type
- Our running example from *Objects First* is a music organizer
  - An `ArrayList` holding the names of music tracks
- All library classes are available from  
<https://docs.oracle.com/en/java/javase/11/docs/api/>

**Module** `java.base`

**Package** `java.util`

**Class `ArrayList<E>`**

`java.lang.Object`

`java.util.AbstractCollection<E>`

`java.util.AbstractList<E>`

`java.util.ArrayList<E>`

**Type Parameters:**

`E` - the type of elements in this list

# Fields of the MusicOrganizer

```
import java.util.ArrayList;

public class MusicOrganizer
{
    // Storage for an arbitrary number of file names.
    private ArrayList<String> files;

    /**
     * Perform any initialization required for the
     * organizer.
     */
    public MusicOrganizer()
    {
        files = new ArrayList<>();
    }

    ...
}
```

# Dissection – import the library class

```
import java.util.ArrayList;
```

- The **import statement** tells Java that we will use the class `java.util.ArrayList` in `MusicOrganizer`
  - In practical terms, it allows us to use the shorthand `ArrayList` to invoke this class

# Dissection – declare the variable

```
private ArrayList<String> files;
```

- The field declaration creates a variable called `files` that can point to an object of type `ArrayList<String>`
  - This is an `ArrayList` that holds objects of type `String`
- One variable allows us to access the entire collection
  - Like all object variables, this variable is initialised to `null`
- So e.g. with other recent examples, you might have

```
private ArrayList<TicketMachine> machines;  
private ArrayList<Student> cits1001;  
private ArrayList<ClockDisplay> timers;
```

# Dissection – create the collection

```
files = new ArrayList<>();
```

- The assignment in the constructor creates an object of type `ArrayList<String>` that is initially empty, and makes `files` point to that object
  - Here the type of the objects to be held on the `ArrayList` is inferred from the type of the variable `files`
  - It is acceptable to restate the type if you want

```
files = new ArrayList<String>();
```

- Note the keyword `new`
  - If you haven't called `new`, you haven't created an object!

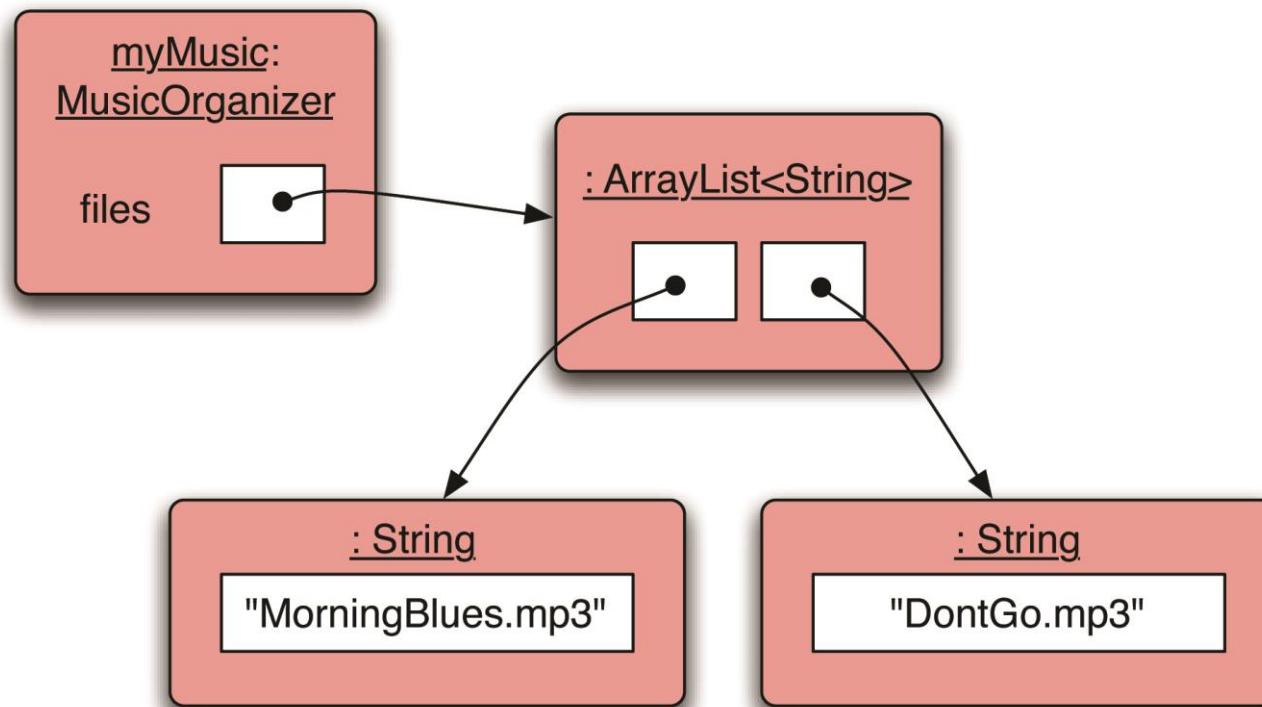
# Operations on ArrayList – add

- The first operation we need is the ability to add an item to the collection
- `files.add(S)`, where `S` can be any expression of the appropriate type
  - In our case, the type `String`
- This adds the object `S` to the end of the collection

# Operations on ArrayList – add

- If we execute the two operations

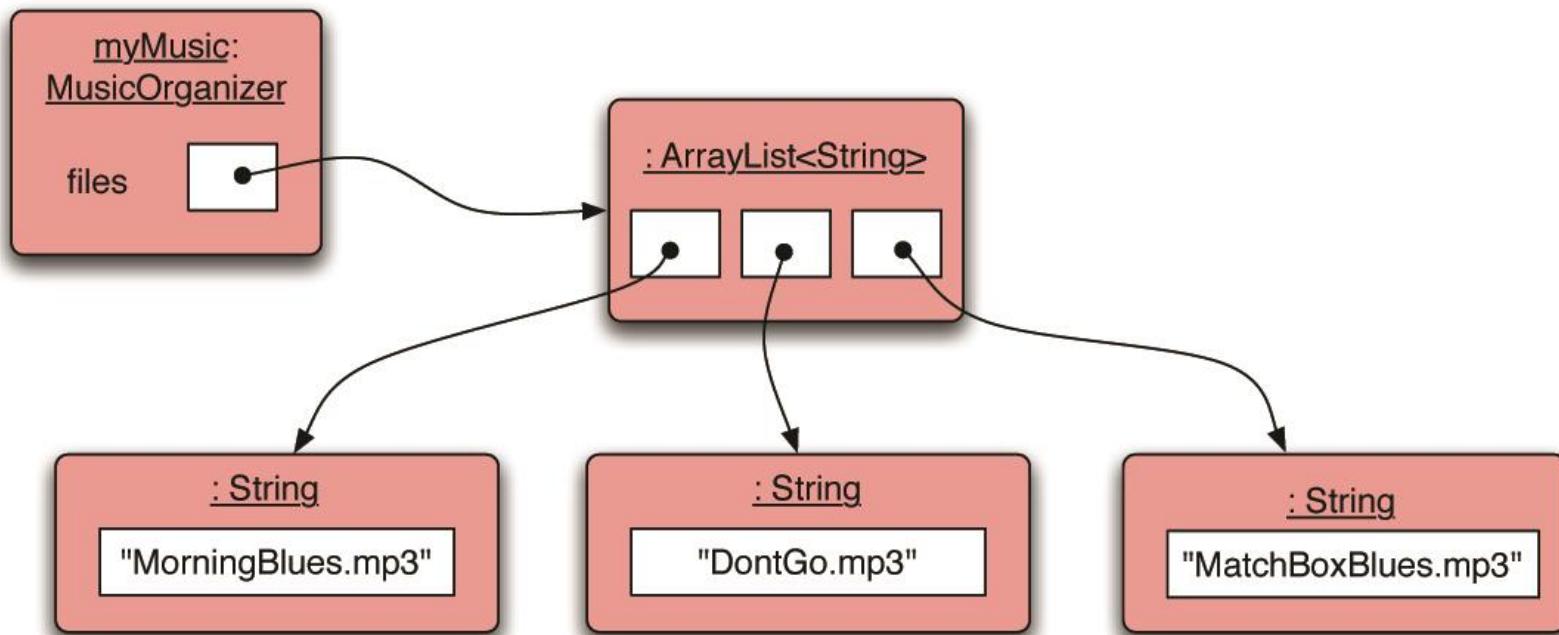
```
files.add("MorningBlues.mp3");  
files.add("DontGo" + "." + "mp3");
```



# Operations on ArrayList – add

- If we then execute the operation

```
files.add("MatchBoxBlues.mp3");
```

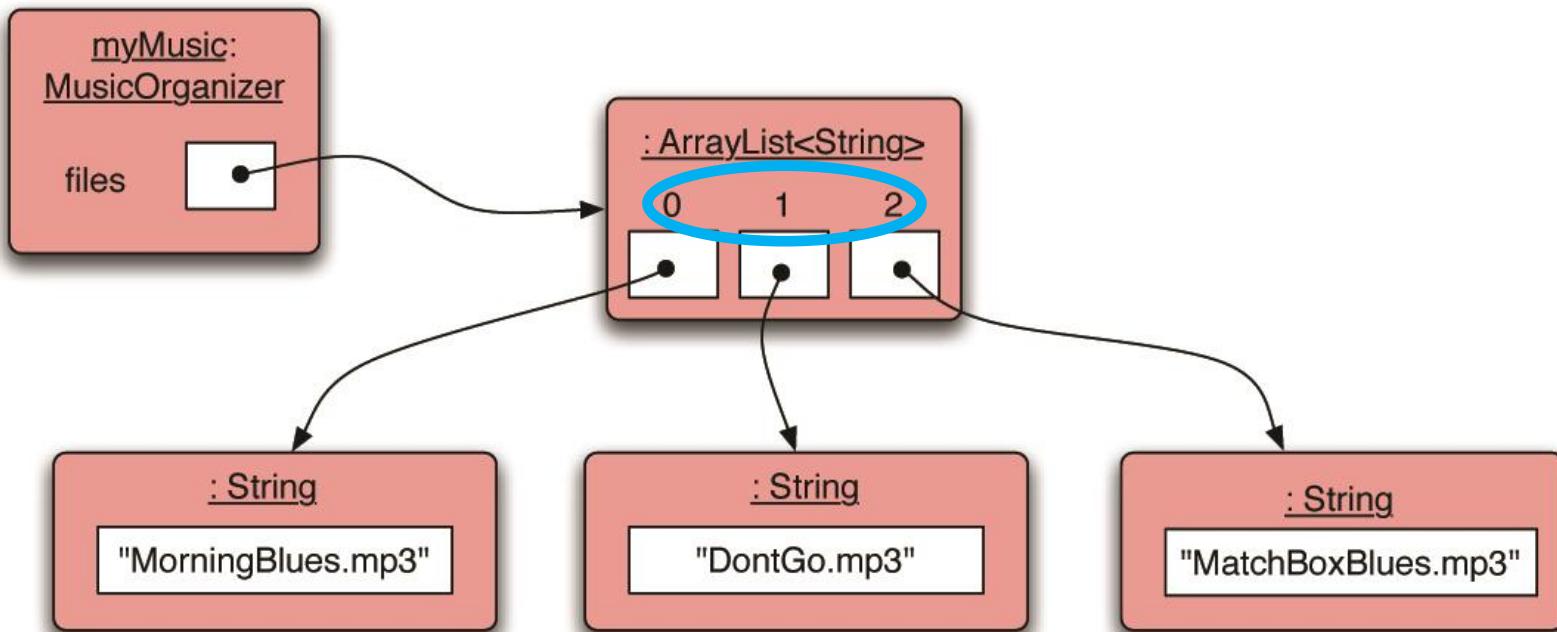


# Operations on ArrayList – size

- We need the ability to ask how many items are currently in the collection
- `files.size()` returns an `int` with this value
- So in the running example, `files.size()` would return 3
- If we call `files.size()` with an empty collection (e.g. immediately after it is created), it returns 0

# Operations on ArrayList – indexing

- We need the ability to retrieve an individual item from the collection
- This is done via *indexing*
- Each item currently in the collection is numbered from 0

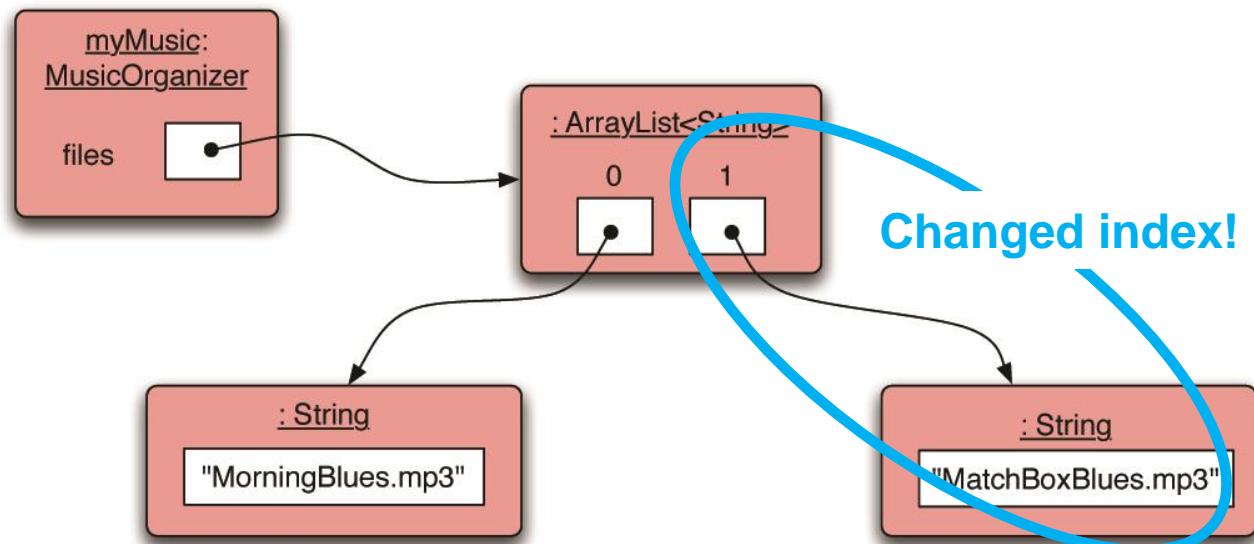


# Operations on ArrayList – get

- To retrieve an item, we need to supply its index
- `files.get(k)` returns the  $k^{\text{th}}$  item in the current collection
- So in the running example
  - `files.get(0)` would return "MorningBlues.mp3"
  - `files.get(1)` would return "DontGo.mp3"
  - `files.get(2)` would return "MatchBoxBlues.mp3"
- Legal arguments to `files.get` are in the range  
 $0 \dots \text{files.size()}-1$ 
  - For whatever size the collection has *at that time*
- If `files.get` is called with an argument outside this range, the program will crash immediately
  - With an `IndexOutOfBoundsException` error message
  - So be careful!

# Operations on ArrayList – remove

- We need the ability to remove an item from the collection
- `files.remove(k)` removes the  $k^{\text{th}}$  item
- The argument must be in the range  $0 \dots \text{files.size()}-1$ 
  - Or the program will crash
- Note also that `remove` will change the indices of some items
  - e.g. after `files.remove(1)`



# Methods of the MusicOrganizer

```
public void addFile(String filename) {  
    files.add(filename);  
}  
  
public int getNumberOfFiles() {  
    return files.size();  
}  
  
public void listFile(int index) {  
    if (0 <= index && index < getNumberOfFiles())  
        System.out.println(files.get(index));  
}  
  
public void removeFile(int index) {  
    if (0 <= index && index < getNumberOfFiles())  
        files.remove(index);  
}
```

The diagram illustrates the classification of methods based on their functionality:

- accessor method**: Points to the `getNumberOfFiles()` method.
- mutator methods**: Points to the `addFile()` and `removeFile()` methods.
- protected accesses**: Points to the `listFile()` method.

# Operations on ArrayList – others

- The library class provides many other operations on ArrayLists, e.g.
  - Adding an object at a specific index
  - Adding multiple objects at the same time
  - Updating the item at a specific index
  - Removing a specific object
  - Removing multiple items, or all items
  - Searching for a specific object
- Have a look!
  - You need to be able to make sense of these libraries
  - If you are unsure, ask!
- Do we need to know *how* all these operations work?
  - No – we just need to know how to use them
  - Abstraction again!

# An aside – what about primitive types?

- ArrayLists can store only object types
- What if we want e.g. a collection of int, or of boolean?
- We can't do this directly, but the Java libraries provide an object type equivalent to each primitive type
  - e.g. Integer for int, Boolean for boolean, Double for double, etc.
- So this is **illegal**

```
ArrayList<int> xs;
```

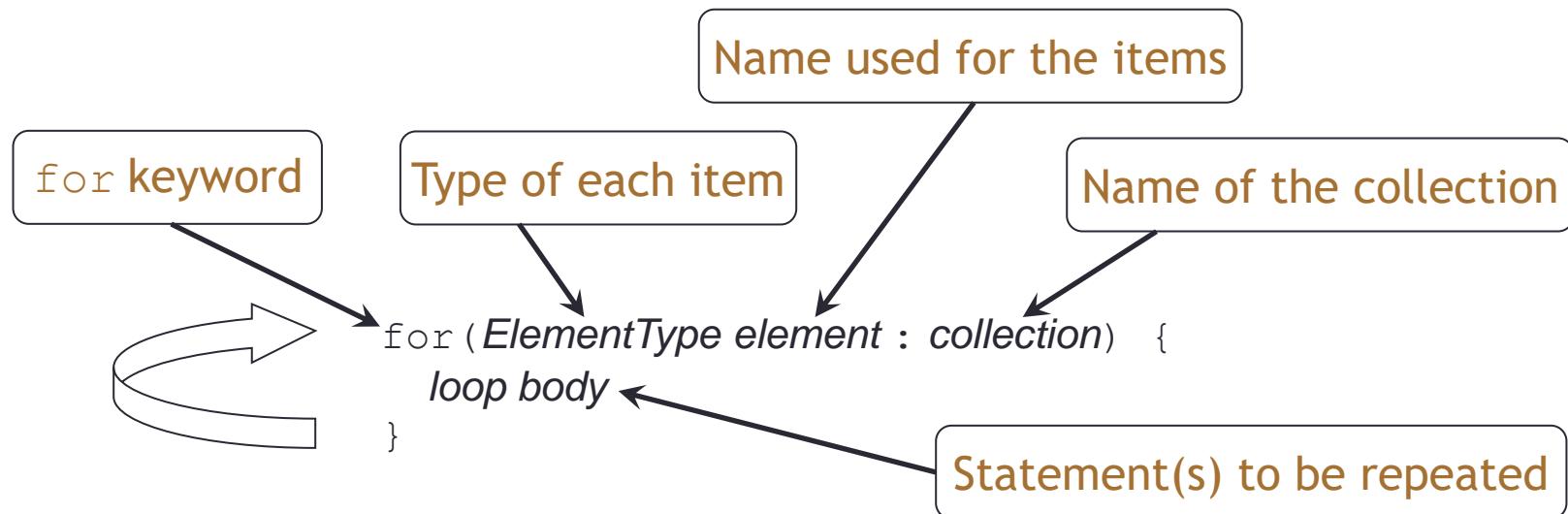
- But this is legal

```
ArrayList<Integer> xs;  
xs = new ArrayList<>();  
xs.add(27);  
int z = xs.size();           // sets z to 1  
boolean b = xs.get(0) == 27; // sets b to true
```

# Processing the items in a collection

- Often we want to apply the same processing individually to all of the items in a collection
  - e.g. change all music tracks from MP3 to MP4
  - e.g. reset all ticket machines on the system to 0
  - e.g. renew all student enrolments in the database
- We want to say something like  
“for every item  $F$  in `files`, process  $F$ ”
- For this situation, Java provides the *for-each* loop
- This is our first example of *iteration*
  - Repeating actions to achieve a task
  - Lots more to come on this!

# For-each loop template



For each *element* in *collection*, do the things in the *loop body*.

# An example – list all tracks in the collection

```
public void listAllFiles()  
{  
    for (String filename : files)  
        {System.out.println(filename);}  
}
```

- If there are  $n$  items in the `ArrayList` `files`, this loop will perform  $n$  iterations
  - i.e. the body will be executed  $n$  times
- In the first iteration, `filename` is set to the first item, and the loop body is executed
- In the second iteration, it is set to the second item, and the loop body is executed again
- In the third iteration, it is set to the third item, and the loop body is executed again
- etc.

# Sometimes not all items are wanted

- So use a conditional in the body!
- This method lists only tracks whose names contains searchString
  - contains is a method from the String library class
  - Look it up!
  - e.g. if filename = "Sydney"; filename.contains("ne") returns true, but filename.contains("sy") returns false

```
public void findFiles(String searchString)
{
    for (String filename : files) {
        if (filename.contains(searchString))
            {System.out.println(filename); }
    }
}
```

# Sometimes we want only the first matching item

- This pattern is known as *search and return*
- If we find a match, the first return is executed
  - And the method terminates at that point
- If no match is found, the second return is executed
  - After all items have been tried

```
public String findFile(String searchString)
{
    for (String filename : files) {
        if (filename.contains(searchString)) {
            return filename; // return the first match
        } // if one is found
    }
    return ""; // return the empty string
} // if NO match is found
```

That's all for this lecture

# CITS1001

## 8. REPETITION – PART 1

---

*Objects First With Java,*  
Chapter 4 and Chapter 7

# Lecture essentials

- Repetition
  - Repeating the same set of actions multiple times
- Definite repetition, where we know in advance the number of iterations
  - for and for-each loops

# Iteration fundamentals

- We often want to repeat some actions over and over
  - e.g. “do this action for each student in CITS1001”
  - e.g. “do this action a hundred times”
  - e.g. “do this action until the room is empty”
- Loops provide us with a way to repeat actions, and to control how many times we repeat those actions
- The first paradigm above is done by the *for-each* loop
  - See previous lecture
- The second paradigm is done by the *for* loop
- The third paradigm is done by the *while* loop
  - Also the *do-while* loop

# Templates of the various loops

```
for (Type element : collection) {  
    statements to be repeated  
}
```

---

```
for (initialization; condition; post-body action) {  
    statements to be repeated  
}
```

---

```
while (condition) {  
    statements to be repeated  
}
```

---

```
do {  
    statements to be repeated  
} while (condition);
```

# The for loop

- The for loop is the most frequently-used looping structure

The diagram illustrates the structure of a for loop. It features a horizontal brace that encompasses the initialization, boolean-expression, and post-body update sections of the loop header. Above this brace is a callout box labeled "Header of the loop". Below the brace is another callout box labeled "Body of the loop", which contains the text "statement1; statement2; statement3; etc.". The entire structure is enclosed in a large, thin-lined bracket.

```
for (initialization; boolean-expression; post-body update) {  
    statement1;  
    statement2;  
    statement3;  
    etc.  
}
```

- e.g.

```
for (int x = 0; x < 5; x += 1) {  
    System.out.println("I can count to " + x);  
}
```

# Header – the initialization part

- The initialization part can be any Java statement
- It is performed *once only* when execution first reaches the loop
- It is normally used to initialize a *counter variable*
  - AKA the *index variable*

```
for (int x = 0; x < 5; x += 1) {  
    System.out.println("I can count to " + x);  
}
```

# Header – the Boolean expression

- The Boolean expression controls whether or not the body of the loop is executed
- It is often a *test* on the counter variable
- The expression is evaluated immediately after initialization
  - And at the start of every subsequent iteration
- If its value is `true`, the statements in the body are executed
- If its value is `false`, the loop has finished and the statements in the body are NOT executed
  - Execution continues at the first statement *after* the for loop, if any

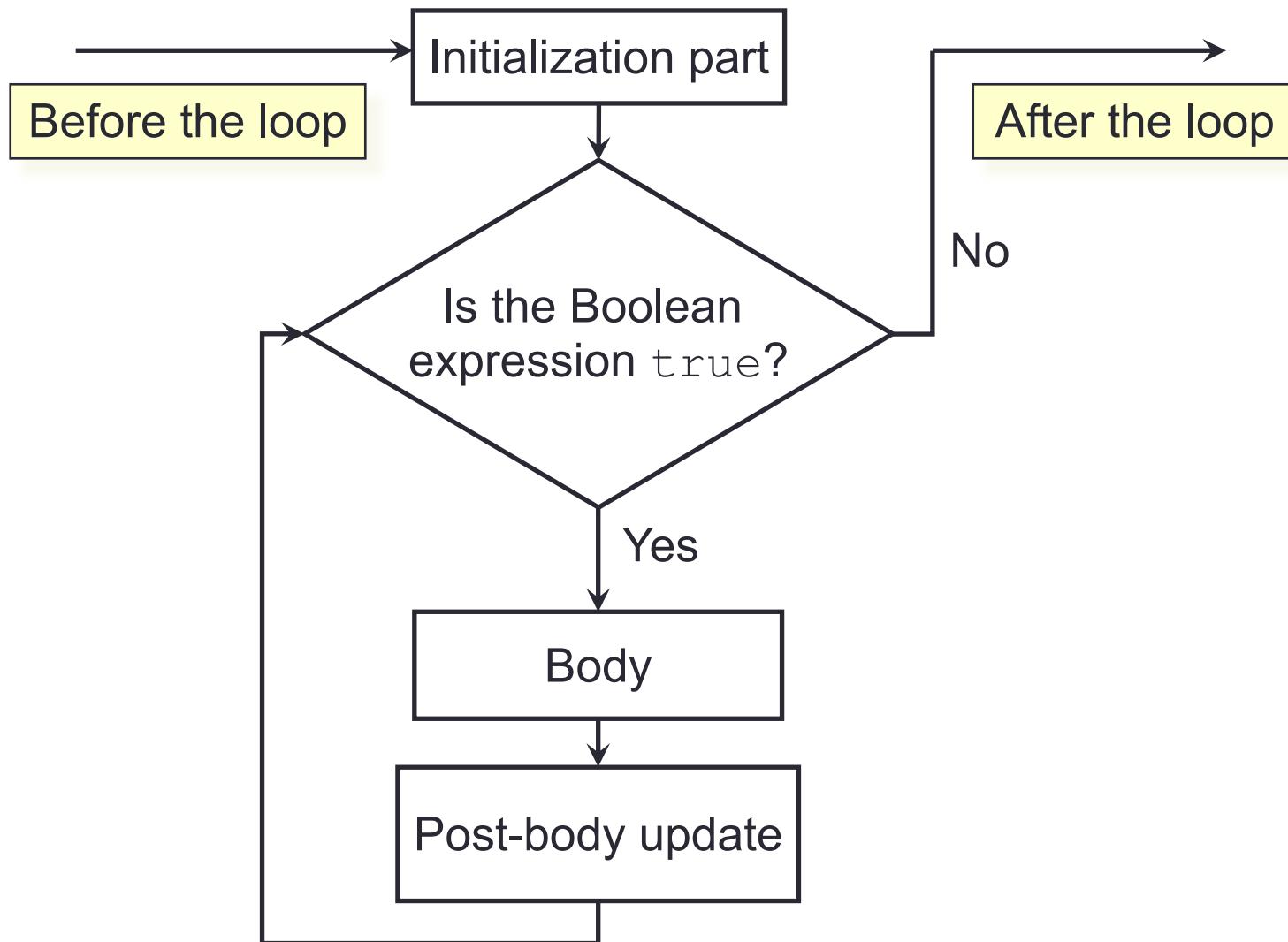
```
for (int x = 0; x < 5 x += 1) {  
    System.out.println("I can count to " + x);  
}
```

# Header – the post-body update

- The post-body update can be any Java statement
- It is performed *each time* the body of the loop is executed
  - Immediately after the last statement of the body has been executed
- It is usually used to *update* the counter variable

```
for (int x = 0; x < 5; x += 1) {  
    System.out.println("I can count to " + x);  
}
```

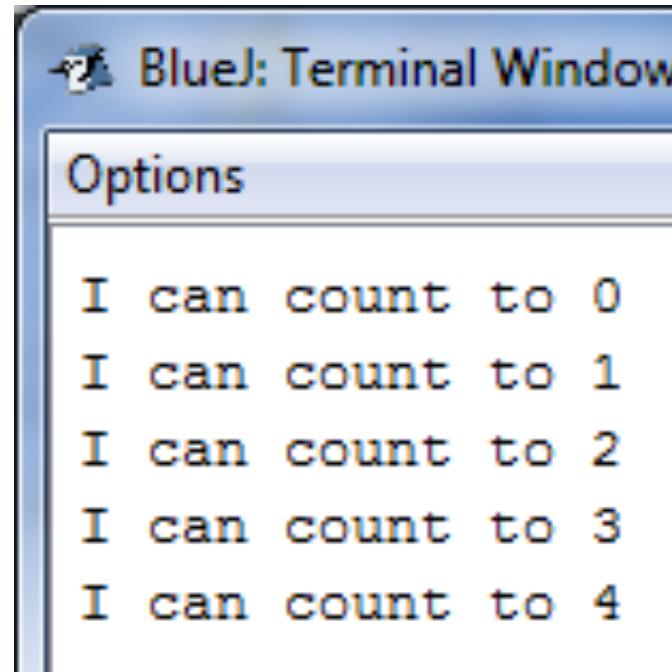
# The for loop flow chart



# The for loop execution

- For loops are most often used when we want to perform a pre-determined number of repetitions

```
for (int x = 0; x < 5; x += 1)
    {System.out.println ("I can count to " + x);}
```



A screenshot of the BlueJ IDE showing a terminal window titled "BlueJ: Terminal Window". The window has a menu bar with "Options" selected. The terminal output displays five lines of text: "I can count to 0", "I can count to 1", "I can count to 2", "I can count to 3", and "I can count to 4".

```
I can count to 0
I can count to 1
I can count to 2
I can count to 3
I can count to 4
```

# How did this work?

- First iteration
  - Initialization declares `x` and sets it to 0
  - `Check if x < 5: returns true`
  - Execute body – sends “I can count to 0” to the terminal window
  - `Add 1 to x: it becomes 1`
- Second iteration
  - `Check if x < 5 : returns true`
  - Execute body – sends “I can count to 1” to the terminal window
  - `Add 1 to x: it becomes 2`
  - `Check if x < 5 : returns true`
  - Execute body – sends “I can count to 2” to the terminal window
  - `Add 1 to x: it becomes 3`
  - `Check if x < 5 : returns true`
  - Execute body – sends “I can count to 3” to the terminal window
  - `Add 1 to x: it becomes 4`
  - `Check if x < 5 : returns true`
  - Execute body – sends “I can count to 4” to the terminal window
  - `Add 1 to x: it becomes 5`
  - `Check if x < 5 : returns false`

```
for (int x = 0; x < 5; x += 1)  
{System.out.println ("I can count to " + x); }
```

# An aside – the increment operator

- The post-body update so often consists of “adding 1” that Java provides a short-hand notation for this operation
- `i += 1` (equivalent to `i = i+1`) may be replaced by `i++`

```
for (int i = 0; i < 5; i++) {  
    System.out.println("I can count to " + i);  
}
```

- In fact, `i++` is both a *statement* and an *expression*
  - As an expression, it has the value of `i` before the incrementing

**Warning:** Use either `i += 1;` or `i++;`  
but *never* use them together as `i = i++;`

# A one-statement body

- If the body of the loop contains only one statement, it is legal to omit the {}, e.g.

```
for (int i = 0; i < 5; i++) {  
    System.out.println("I can count to " + i);  
}
```

is equivalent to

```
for (int i = 0; i < 5; i++)  
    System.out.println("I can count to " + i);
```

- However, the first form is preferable
  - It clarifies explicitly what is inside the loop body and what is outside
  - It is more robust to code-changes that may be applied later
  - Where we use the second form on slides, that is only to save space!

# An example – summing numbers

- The method `sum` sums the numbers between 1 and `n`, so e.g. `sum(6)` returns 21

```
private int sum(int n) {  
    int z = 0; // initialize z  
    for (int i = 1; i <= n; i++)  
        {z += i;} // each iteration adds one number to z  
    return z; // calculate z  
}
```

- `z` is called an *accumulating parameter* or *accumulating variable*
  - This is an **extremely common** paradigm
  - The variable gets initialized, then calculated in the loop, then returned

# An example – summing odd numbers

- The method `odds` sums the odd numbers between 1 and `n`, so e.g. `odds(6)` returns 9

```
private int odds(int n) {  
    int z = 0;  
    for (int i = 1; i <= n; i++)  
        {if (i % 2 == 1) {z += i;}}}  
    return z;  
}
```

- The conditional picks out the odd numbers
  - Even numbers are not added to `z`

# Equivalently

- The method `odds` sums the odd numbers between 1 and `n`

```
private int odds(int n) {  
    int z = 0;  
    for (int i = 1; i <= n; i += 2)  
        {z += i;}  
    return z;  
}
```

- `i` starts odd, and the post-body update keeps it odd
  - Even numbers never arise at all
- This version is somewhat neater

# Equivalently

- The method `odds` sums the odd numbers between 1 and  $n$

```
private int odds(int n) {  
    int z = 0;  
    for (int i = 1; i <= (n+1)/2; i++)  
        {z += 2*i-1;}  
    return z;  
}
```

- The RHS of the assignment creates the odd numbers
- This version is somewhat more complicated
  - But make sure you understand that it always returns the same result as the earlier versions

# An aside – tracing code

- To be sure we understand how the last version of odds operates, we can *trace* how an example actually works
  - Consider odds(8), so  $n = 8$  and doesn't change ever
  - Build a table that tracks how the other variables change their values

```
int z = 0;
for (int i = 1;
     i <= (n+1)/2;
     i++)
{
    z += 2*i-1;
}
```

**one iteration  
of the loop**

<b>z</b>	<b>i</b>	
0	1	$i \leq 4?$
1	2	$i \leq 4?$
4	3	$i \leq 4?$
9	4	$i \leq 4?$
16	5	$i \leq 4?$

- It sometimes takes tracing multiple examples to get a proper understand of how a code fragment works

# Making tables

- Another common use of for loops is making tables or rows of information, e.g.

Celsius	Fahrenheit
0	32
5	41
10	50
15	59
...	
95	203
100	212

# Making tables

- The method temps prints a table of temperature conversions from Celsius to Fahrenheit

```
private void temps()
{
    System.out.println("Celsius\tFahrenheit");
    for (int c = 0; c <= 100; c += 5) {
        System.out.println(c + "\t" + (c * 9 / 5 + 32));
    }
}
```

- Note the careful bracketing in the print statement, and the use of the tab character \t

# Building data structures

- Another common use is building data structures
- The method `numbers` returns an `ArrayList` containing the natural numbers up to `n`

```
private ArrayList<Integer> numbers(int n)
{
    ArrayList<Integer> z = new ArrayList<>();
    for (int k = 0; k <= n; k++) { z.add(k); }
    return z;
}
```

- The accumulating variable works in the same way as earlier
  - Even though it represents a data structure

# A numerical example

- The value of  $\pi$  can be specified exactly by the formula

$$\pi = 4(1 - 1/3 + 1/5 - 1/7 + 1/9 - \dots)$$

- There are two ways we might use this to approximate  $\pi$ 
  - Sum a pre-determined number of terms
  - Sum terms until we achieve a pre-determined precision
- The first is best done using a for loop
- The second is best done using a while loop (see later)

# $\pi$ using a given number of terms

- The method `pil` sums  $n$  terms to approximate  $\pi$

```
private double pil(int n)
{
    double approx = 0;
    double mult = 4;
    for (int i = 0; i < n; i++) {
        approx += mult / (2*i+1);
        mult = -mult;
    }
    return approx;
}
```

**n iterations**

**accumulating variable**

**terms alternate +ve and -ve**

```
graph TD; n((n)) --> i((i)); approx((approx)) --> approxLine[approx += mult / (2*i+1)]; mult((mult)) --> multLine[mult = -mult]; approxLine --> return((return approx));
```

# Situation at the top of the loop

i	mult	2*i+1	approx
0	4.0	1	0.00
1	-4.0	3	4.00
2	4.0	5	2.67
3	-4.0	7	3.47
4	4.0	9	2.90
5	-4.0	11	3.34
6	4.0	13	2.98

Loop stops when  $i$  reaches  $n$

Continued in Part 2

# CITS1001

## 8. REPETITION – PART 2

---

*Objects First With Java,*  
Chapter 4

# Lecture essentials

- for vs for-each loops
- Indefinite repetition, where we don't know in advance the number of iterations
  - while and do-while loops

# For loops vs. for-each loops

- Any for-each loop can be written as a for loop

```
for (Type elem : coll)
{
    statements;
}
```

```
for (int k = 0; k < coll.size(); k++)
{
    Type elem = coll.get(k);
    statements;
}
```

- The for-each loop on the left says
  - For each item `elem` in collection `coll`,  
do `statements`
- The for loop on the right says
  - For each legal index `k` into collection `coll`,  
set `elem` to the  $k^{\text{th}}$  item and do `statements`

# For loops vs. for-each loops

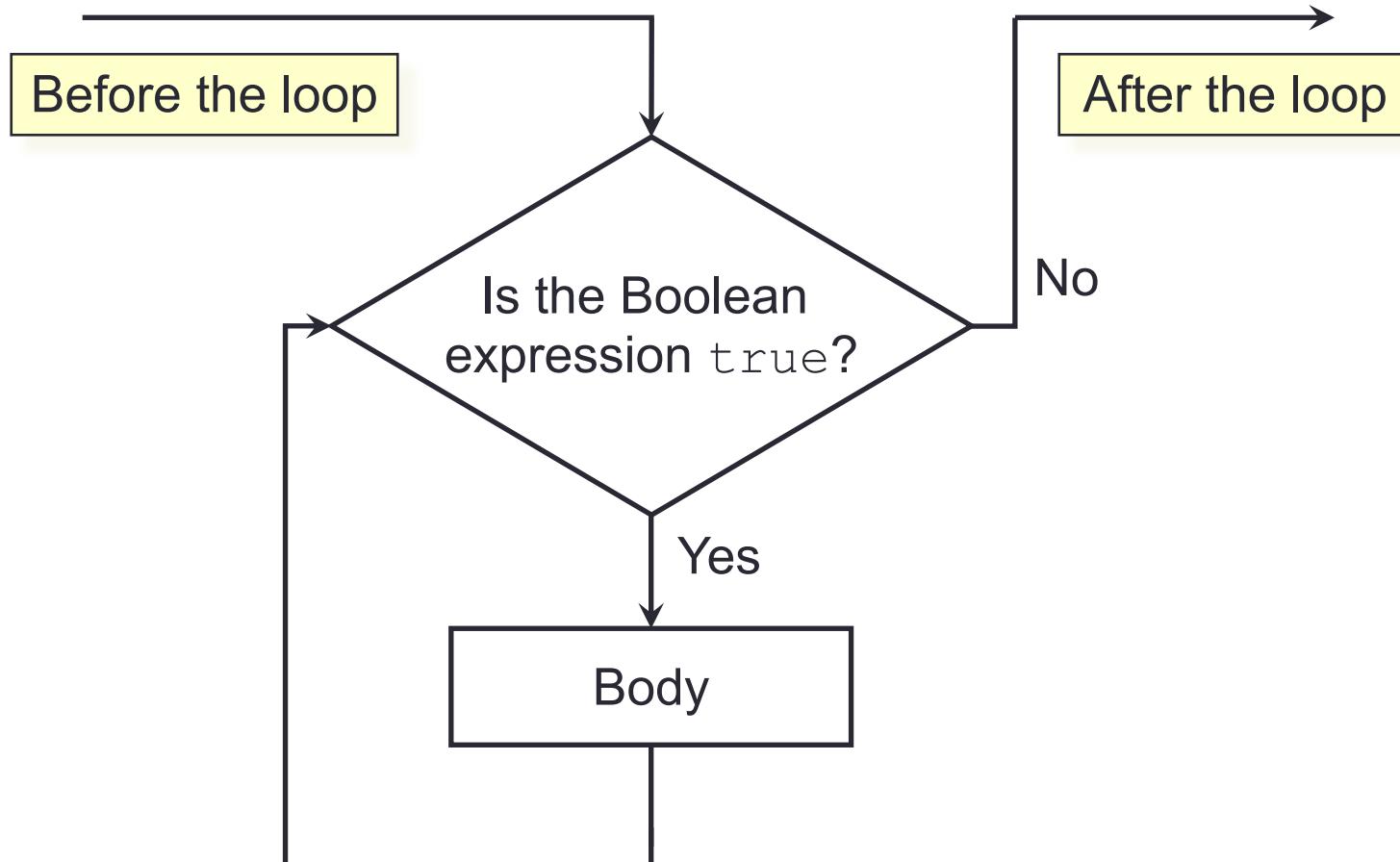
- For loops are more general than for-each loops
  - e.g. the latter apply only when processing collections
- Even with collections, anything that requires indexing needs a for loop
  - Or anything requiring use of the step-size
- But for-each is usually preferable where applicable
  - They make it explicit that a collection is being processed
  - They are somewhat less error-prone, as they involve less detail

# The while loop

- The third paradigm for repetition is “do this action until this condition becomes true”
  - We don’t know in advance how many iterations there will be
  - Including the possibility there might be zero iterations
- In Java this is usually done with a while loop
- We use a Boolean condition to decide whether or not to keep iterating

```
while (condition) {  
    statements to be repeated  
}
```

# The while loop flow chart



# Looking for your keys

```
while(the keys are missing) {  
    look in the next place  
}
```

- Or equivalently

```
while(not (the keys have been found)) {  
    look in the next place  
}
```

# Two operational differences

- Conceptually, while loops are simpler than for loops
  - They have fewer components
- But there are two operational differences between for and while which can cause problems
- Any initialization needed must be done before the loop
  - Whereas a for loop has explicit initialization in the header
- Termination must be carefully considered
  - It is not hard to accidentally write an “infinite loop” using while
- In BlueJ you can stop an infinite loop with Ctrl-Shift-R

# Looking for your keys

```
boolean searching = true; // used to control the loop  
Location place = firstPlace;  
while(searching) {  
    if(the keys are at place) searching = false;  
    else place = next(place);  
}
```

- But suppose we never find them?
  - People know to give up eventually
  - Computers don't!

# Finding prime numbers

- The method `nextPrime` returns the smallest prime number that is bigger than `n`

```
private int nextPrime(int n) {  
    int z = n + 1;  
    while (!isPrime(z)) z++; // we know there'll be one!  
    return z;  
}
```

```
private boolean isPrime(int k) {  
    for (int i = 2; i < k; i++)  
        {if (k % i == 0) return false;}  
    return true;  
}
```

# Searching a collection

**keep searching iff both  
we haven't found it yet  
we haven't checked  
every item yet**

```
int index = 0;  
boolean found = false;  
while(!found && index < files.size()) {  
    String file = files.get(index);  
    if(file.contains(searchString)) {  
        found = true; // we can stop looking  
    }  
    else {  
        index++; // we should try the next one  
    }  
}  
// Either we found searchString at index;  
// or we searched the whole collection,  
// in which case found = false and index = files.size()
```

# $\pi$ to within a given accuracy

- The method `pi2` sums terms to approximate  $\pi$  until the next term is smaller than accuracy

```

private double pi2(double accuracy) {
    double approx = 0;
    double mult = 4;
    double denom = 1;
    while (Math.abs(mult / denom) > accuracy) {
        approx += mult / denom;
        mult = -mult;
        denom += 2;
    }
    return approx;
}

```

**iterate until this is false**

**accumulating variable**

**terms alternate +ve and -ve**

# Situation at the top of the loop

Step	denom	mult/denom	approx
0	1.0	4.00	0.00
1	3.0	-1.33	4.00
2	5.0	0.80	2.67
3	7.0	-0.57	3.47
4	9.0	0.44	2.90
5	11.0	-0.36	3.34
6	13.0	0.31	2.98



Loop stops when the absolute size of  
the next term is less than accuracy

# For loops vs. while loops

- Any for loop can be written as a while loop

```
for (s1; b; s2)
{
    statements;
}
```

```
s1;
while (b)
{
    statements;
    s2;
}
```

- The for loop on the left says
  - Do s1; then if b is true, do statements and s2, then iterate again
- After s1, the while loop on the right says
  - If b is true, do statements and s2, then iterate again

# For loops vs. while loops

- For loops are generally preferable to while loops
  - Easier for human readers to count the number of iterations
  - Less prone to infinite looping
- But while loops are more general
  - They allow any type of stopping condition

# The do-while loop

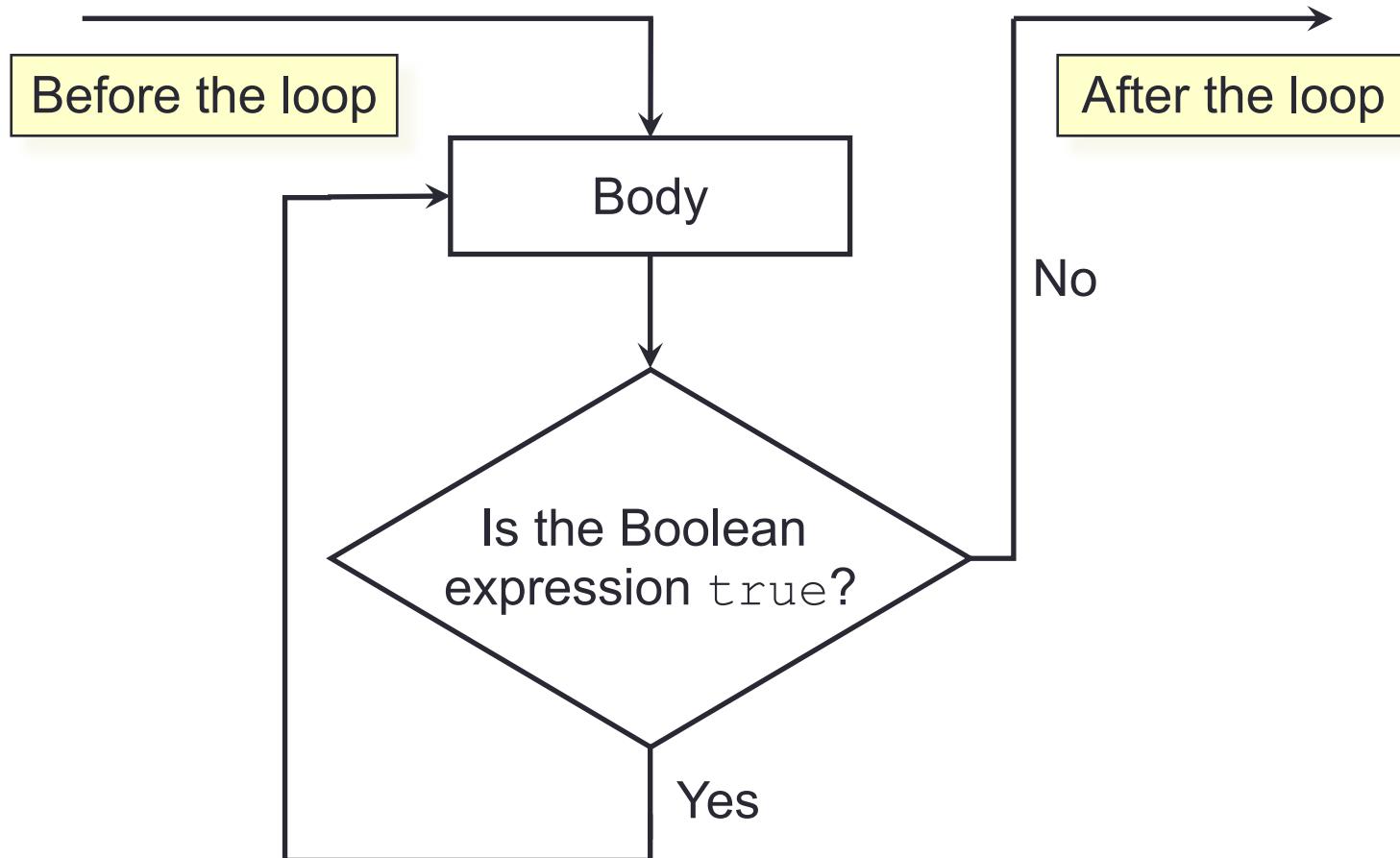
- The do-while loop is logically very similar to the while loop
  - It iterates until some condition becomes false
- The only real difference is that do-while guarantees to perform at least one iteration
- So these two are equivalent

```
body  
while (condition)  
  {body}
```

```
do  
  {body}  
  while (condition);
```

- do-while also is susceptible to improper initialization and improper termination

# The do-while loop flow chart



# Asking for a password

- The method `allowEntry` allows a user into a system if they enter a valid password

```
private boolean allowEntry(User user)
{
    boolean success;
    do {
        String attempt = // read password
        success = user.matchPassword(attempt);
    } while (!success);
    return success;
}
```

# Asking for a password

- Hang on, that version is no good!
  - If you don't know the password, you're in trouble
  - There's no way out of the loop

```
private boolean allowEntry(User user)
{
    boolean success = false;
    boolean givenUp = false;
    do {
        String attempt = // read password
        if (attempt == "") {givenUp = true;}
        else {success = user.matchPassword(attempt);}
    } while (!success && !givenUp);
    return success;
}
```

# The End

- Of repetition

# CITS1001

## 9. JUNIT – A BRIEF DEMONSTRATION

---

# Lecture essentials

- A brief demonstration of the practical use of JUnit classes
- Junit tests run your code, and compare actual results with expected results
- Messages give you
  - The line in the testing class where the failed test is
  - The expected result and the returned value
  - A user-defined error message
- Find and fix the error(s), and rerun the tests
- There are often a *lot* of tests
  - Tests are your friends!
  - Using testing classes supports frequent testing
- Remember **passing all tests ≠ perfect program!**

# The naïve ticket machine class

- `printTicket` has at least three shortcomings
  - `balance` is updated wrongly
  - `total` is updated wrongly
  - It issues tickets without checking the balance first

```
public void printTicket() {  
    // Simulate the printing of a ticket.  
    System.out.println("#####"); // printing omitted  
  
    // Update the total collected with the balance.  
    total += balance;  
    // Clear the balance.  
    balance = 0;  
}
```

- How should we test it?

# An example JUnit testing method

- The field `int price = 500;`

```
@Test
public void testprintTicket() {
    String m = "printTicket";
    tm.insertMoney(500);
    assertEquals(error(1, m), 500, tm.getBalance());
    assertEquals(error(2, m), 0, tm.getTotal());
    tm.insertMoney(800);
    assertEquals(error(3, m), 1300, tm.getBalance());
    assertEquals(error(4, m), 0, tm.getTotal());
    tm.printTicket();
    assertEquals(error(5, m), 800, tm.getBalance());
    assertEquals(error(6, m), 500, tm.getTotal());
    tm.printTicket();
    assertEquals(error(7, m), 300, tm.getBalance());
    assertEquals(error(8, m), 1000, tm.getTotal());
    tm.printTicket();
    assertEquals(error(9, m), 300, tm.getBalance());
    assertEquals(error(10, m), 1000, tm.getTotal());
}
```

# An example JUnit testing method

- The field `int price = 500;`

```

    @Test
    public void testprintTicket() {
        String m = "printTicket";
        tm.insertMoney(500);
        assertEquals(error(1, m), 500, tm.getBalance());
        assertEquals(error(2, m), 0, tm.getTotal());
        tm.insertMoney(800);
        assertEquals(error(3, m), 1300, tm.getBalance());
        assertEquals(error(4, m), 0, tm.getTotal());
        tm.printTicket();
        assertEquals(error(5, m), 800, tm.getBalance());
        assertEquals(error(6, m), 500, tm.getTotal());
        tm.printTicket();
        assertEquals(error(7, m), 300, tm.getBalance());
        assertEquals(error(8, m), 1000, tm.getTotal());
        tm.printTicket();
        assertEquals(error(9, m), 300, tm.getBalance());
        assertEquals(error(10, m), 1000, tm.getTotal());
    }

```

**object and method being tested**

**calls to testing methods**

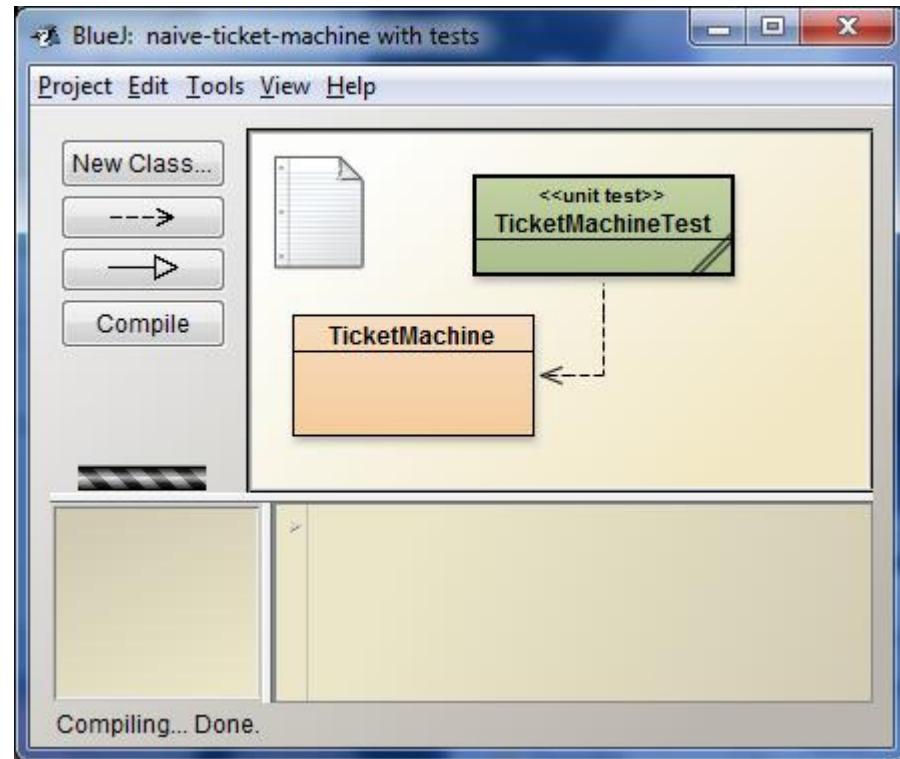
One test

# Built-in testing procedures

- `assertEquals (msg, v, m)`
  - We expect that the method call `m` returns the value `v`
- `assertEquals (msg, v, m, d)`, for doubles
  - We expect that `abs (m - v) < d`,  
i.e. `m` returns something very close to `v`
- `assertTrue (msg, b)`
  - We expect that the expression `b` evaluates to `true`
- `assertFalse (msg, b)`
  - We expect that the expression `b` evaluates to `false`
- When there is a failure, the String `msg` is printed

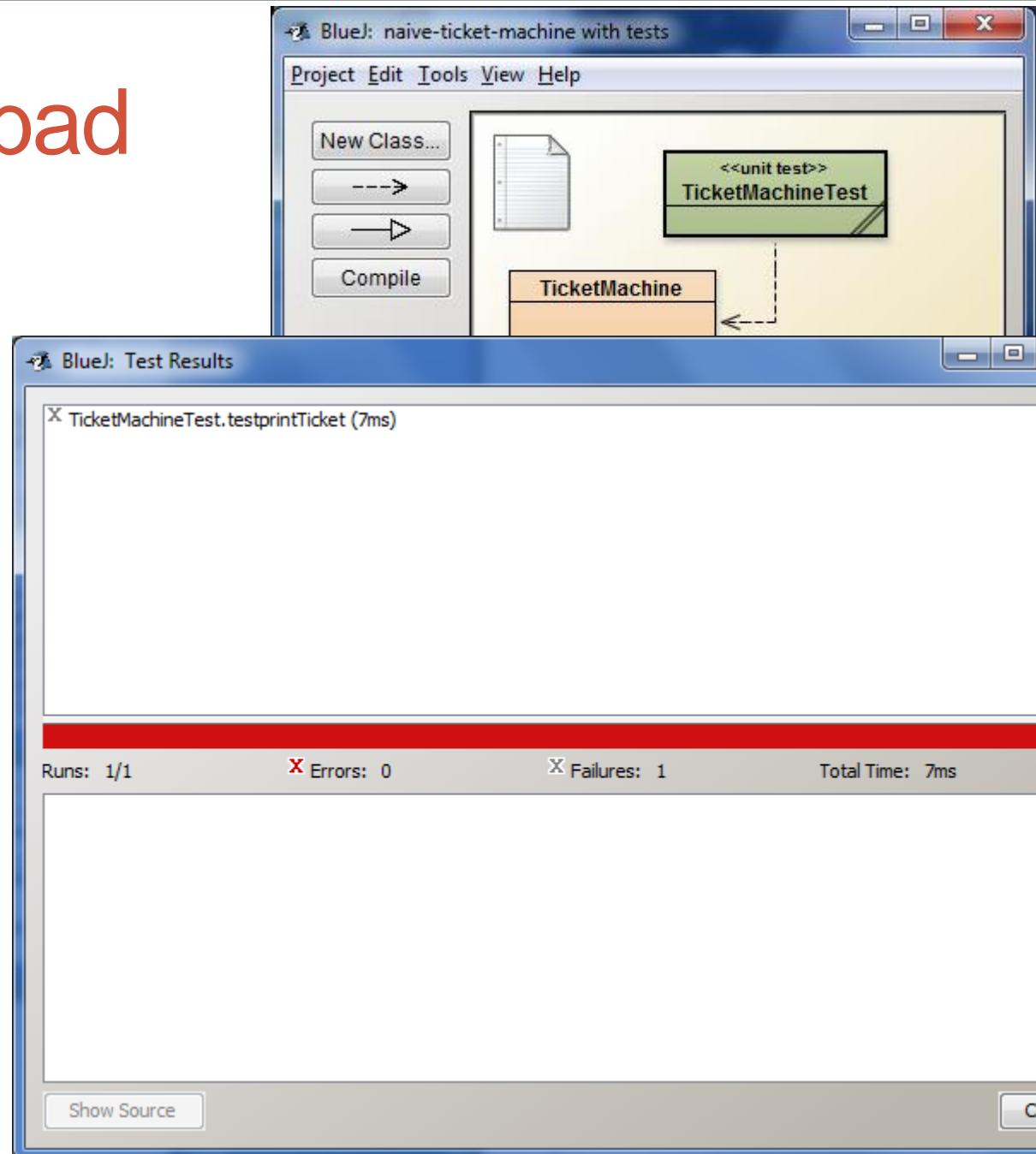
# Operationally

- Right-click on TicketMachineTest and select *Test All*



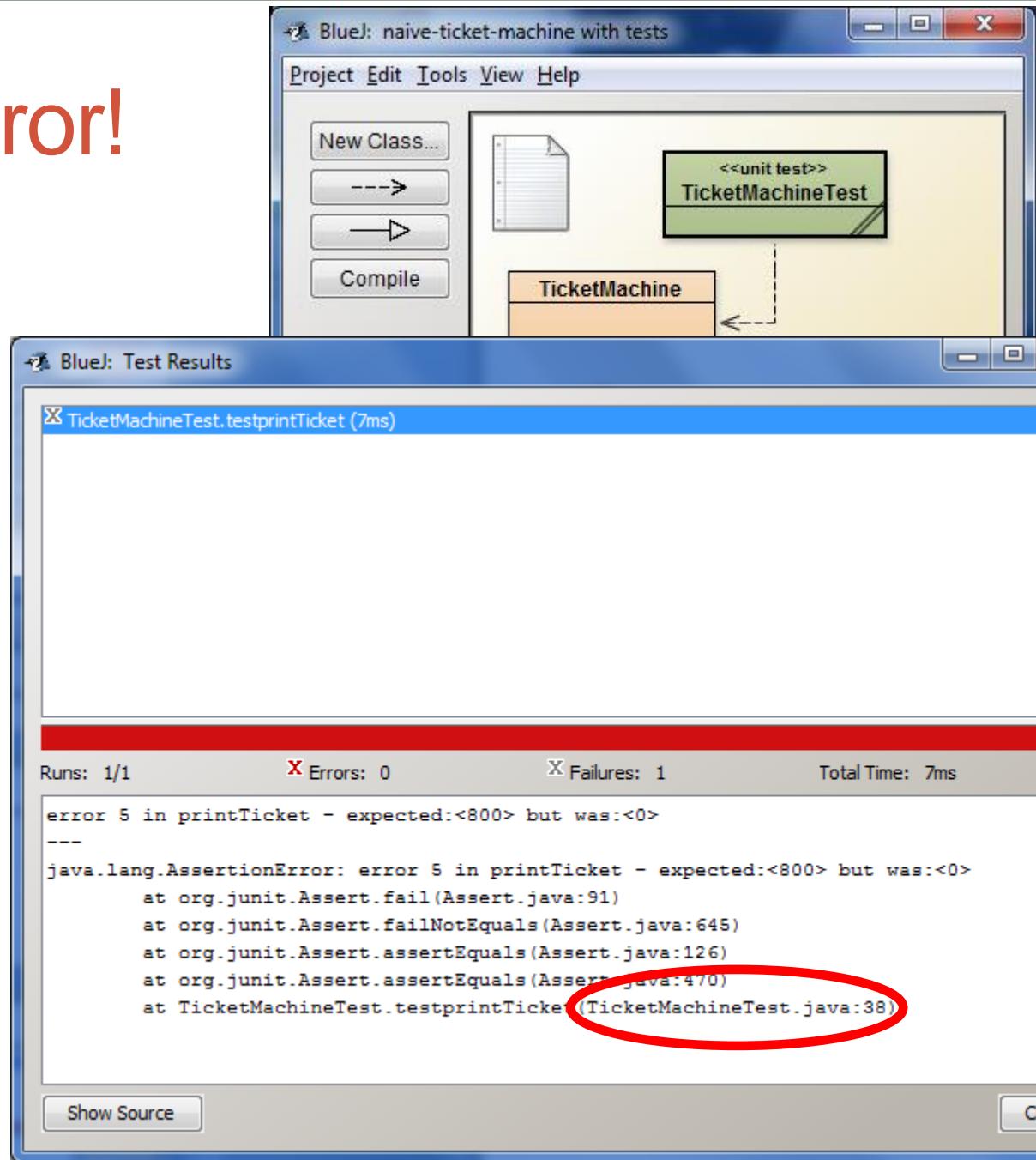
# The cross is bad

- Click on that line



# Locate the error!

- The last line gives the line number at which `testprintTicket` stopped
  - It stops each test method at the first error



# Locate the error!

- The last line gives the line number at which `testprintTicket` stopped
  - It stops each test method at the first error

The screenshot shows the BlueJ IDE interface. At the top, there's a menu bar with Project, Edit, Tools, View, Help. Below the menu is a toolbar with buttons for New Class..., -->, >, and Compile. To the right of the toolbar, there are two class icons: a green one labeled <<unit test>> TicketMachineTest and an orange one labeled TicketMachine. A dashed arrow points from the green class to the orange one. Below the toolbar is a window titled "BlueJ: Test Results". It contains the following Java code:

```
31 tm.insertMoney(500);
32 assertEquals(error(1, m), 500, tm.getBalance());
33 assertEquals(error(2, m), 0, tm.getTotal());
34 tm.insertMoney(800);
35 assertEquals(error(3, m), 1300, tm.getBalance());
36 assertEquals(error(4, m), 0, tm.getTotal());
37 tm.printTicket();
38 assertEquals(error(5, m), 800, tm.getBalance());
39 assertEquals(error(6, m), 500, tm.getTotal());
```

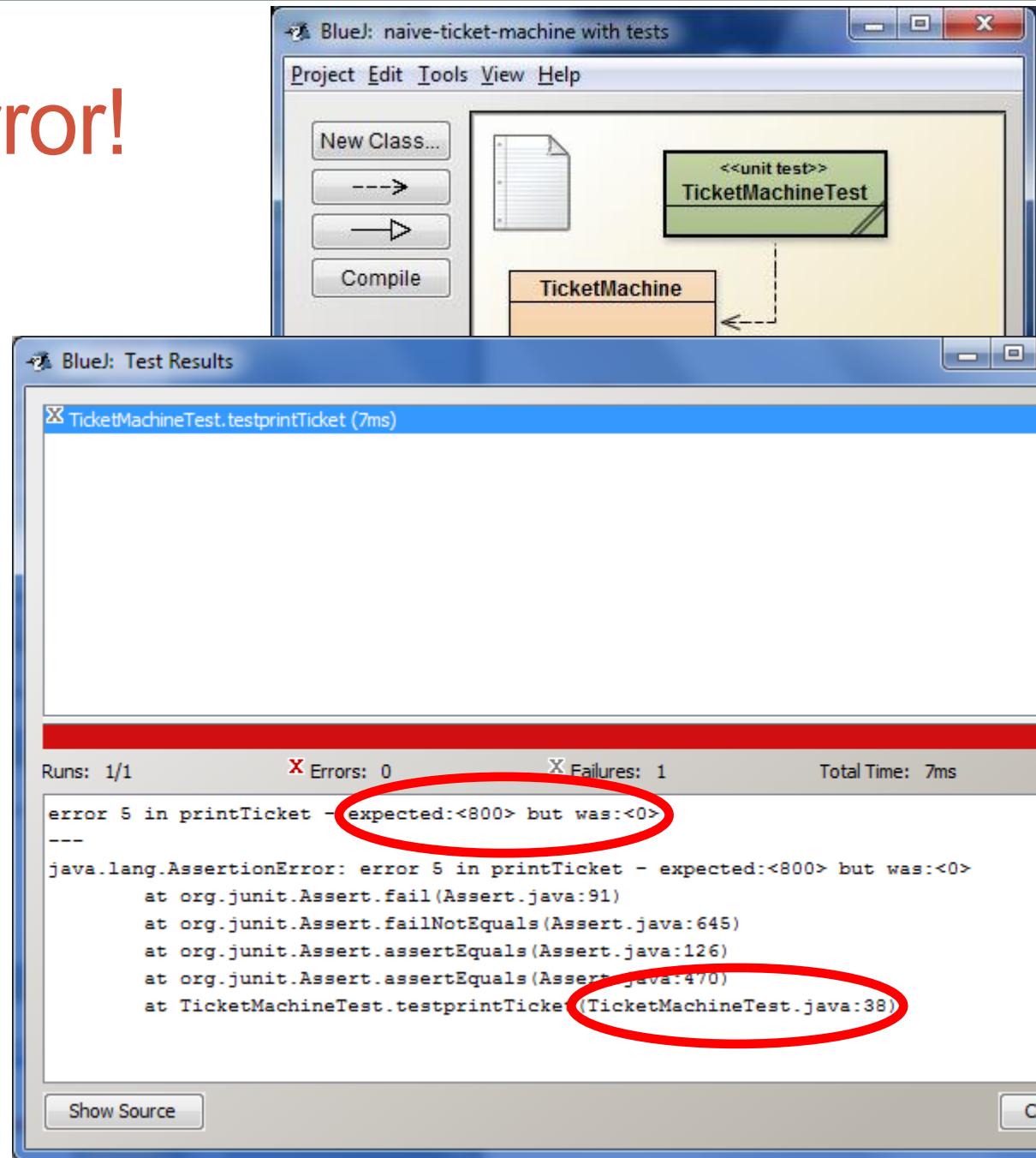
Below the code, the status bar shows: Runs: 1/1, Errors: 0, Failures: 1, Total Time: 7ms. The failure details are listed as:

```
error 5 in printTicket - expected:<800> but was:<0>
---
java.lang.AssertionError: error 5 in printTicket - expected:<800> but was:<0>
    at org.junit.Assert.fail(Assert.java:91)
    at org.junit.Assert.failNotEquals(Assert.java:645)
    at org.junit.Assert.assertEquals(Assert.java:126)
    at org.junit.Assert.assertEquals(Assert.java:470)
    at TicketMachineTest.testprintTicket(TicketMachineTest.java:38)
```

A red oval highlights the line "at TicketMachineTest.testprintTicket(TicketMachineTest.java:38)".

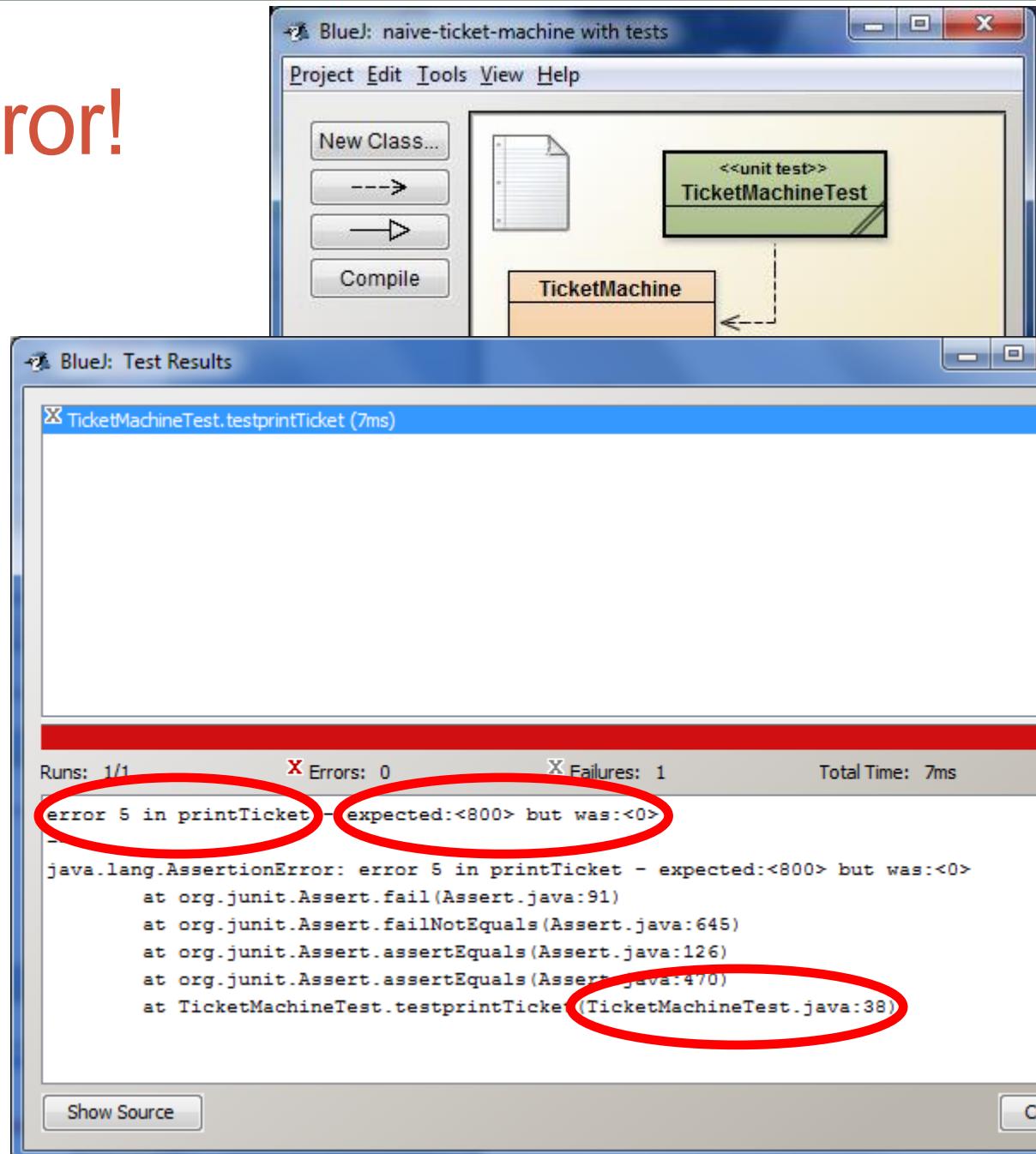
# Locate the error!

- The last line gives the line number at which `testprintTicket` stopped
  - It stops each test method at the first error
- The first line gives
  - The expected result
  - The actual result returned by your method



# Locate the error!

- The last line gives the line number at which `testprintTicket` stopped
  - It stops each test method at the first error
- The first line gives
  - The expected result
  - The actual result returned by your method
- It also prints an error message
  - That may or may not be useful



# So fix the bug!

- balance shouldn't be set to 0

```
public void printTicket() {  
    // Simulate the printing of a ticket.  
    System.out.println("#####"); // printing elided  
  
    // Update the total collected with the balance.  
    total += balance;  
    // Clear the balance.  
    balance = 0;  
}
```

# So fix the bug!

- balance shouldn't be set to 0
  - Just subtract price instead
  - That's how much money was spent...

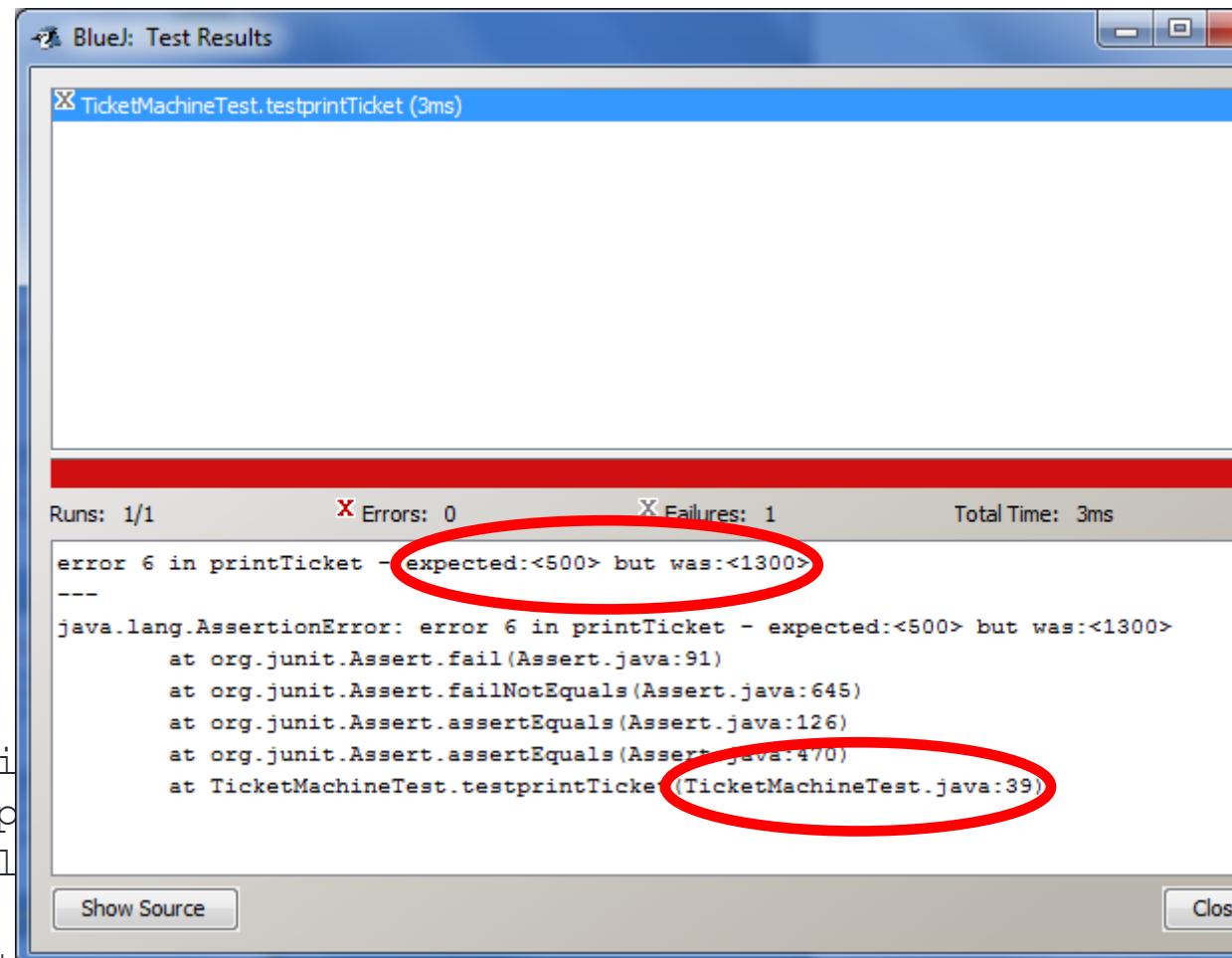
```
public void printTicket() {  
    // Simulate the printing of a ticket.  
    System.out.println("#####"); // printing elided  
  
    // Update the total collected with the balance.  
    total += balance;  
    // Update the balance.  
    balance -= price;  
}
```

# A new error!

- Now it is saying total is wrong

```
public void printTicket() {
    // Simulate the payment.
    System.out.println("Ticket printed");

    // Update the total collected with the balance.
    total += balance;
    // Update the balance.
    balance -= price;
}
```

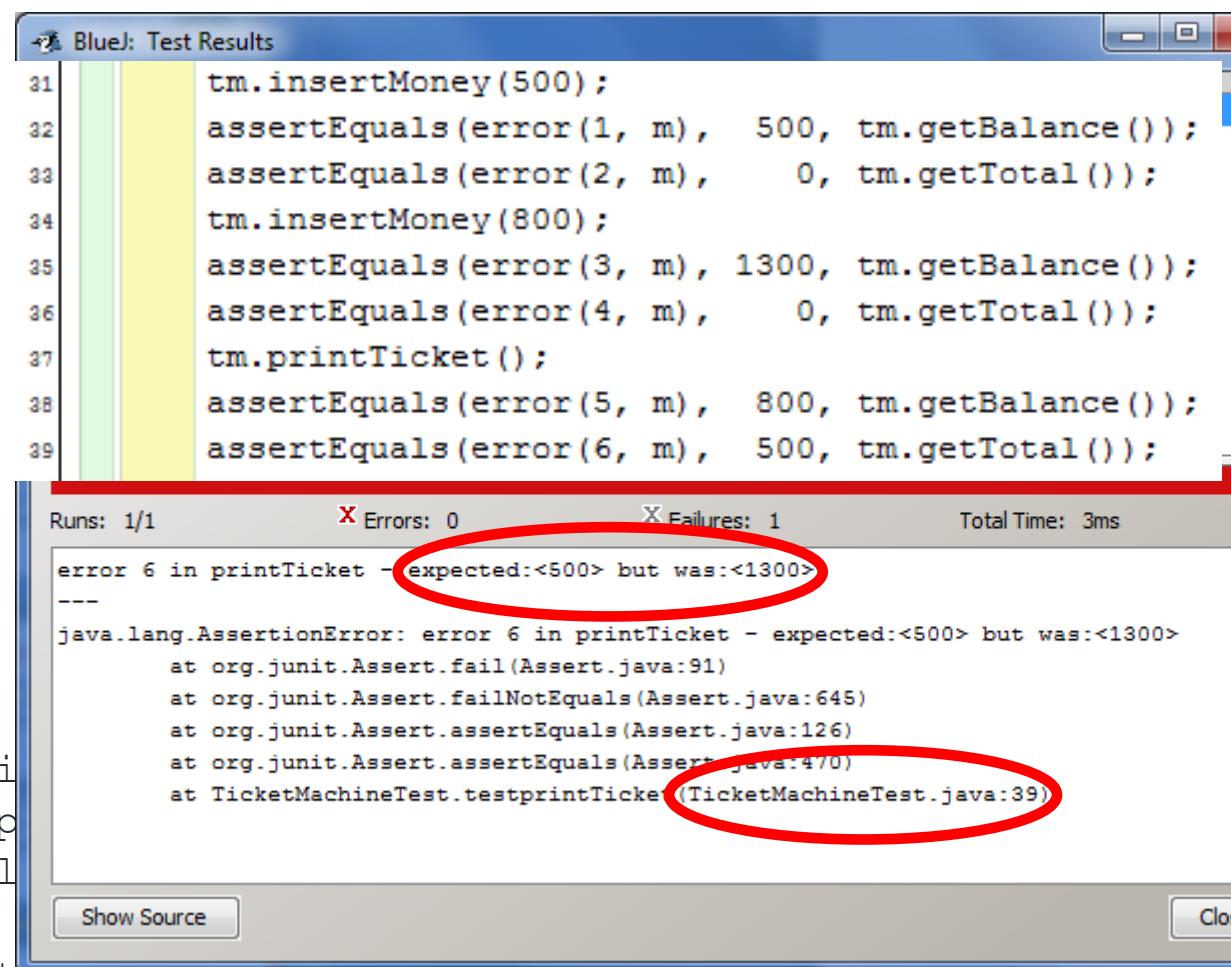


# A new error!

- Now it is saying total is wrong

```
public void printTicket() {
    // Simulate the purchase.
    System.out.println("Ticket printed");

    // Update the total collected with the balance.
    total += balance;
    // Update the balance.
    balance -= price;
}
```



# Fix the second bug

- total shouldn't increase by balance

```
public void printTicket() {  
    // Simulate the printing of a ticket.  
    System.out.println("#####"); // printing elided  
  
    // Update the total collected with the balance.  
    total += balance;  
    // Update the balance.  
    balance -= price;  
}
```

# Fix the second bug

- total shouldn't increase by balance
  - Only price was spent

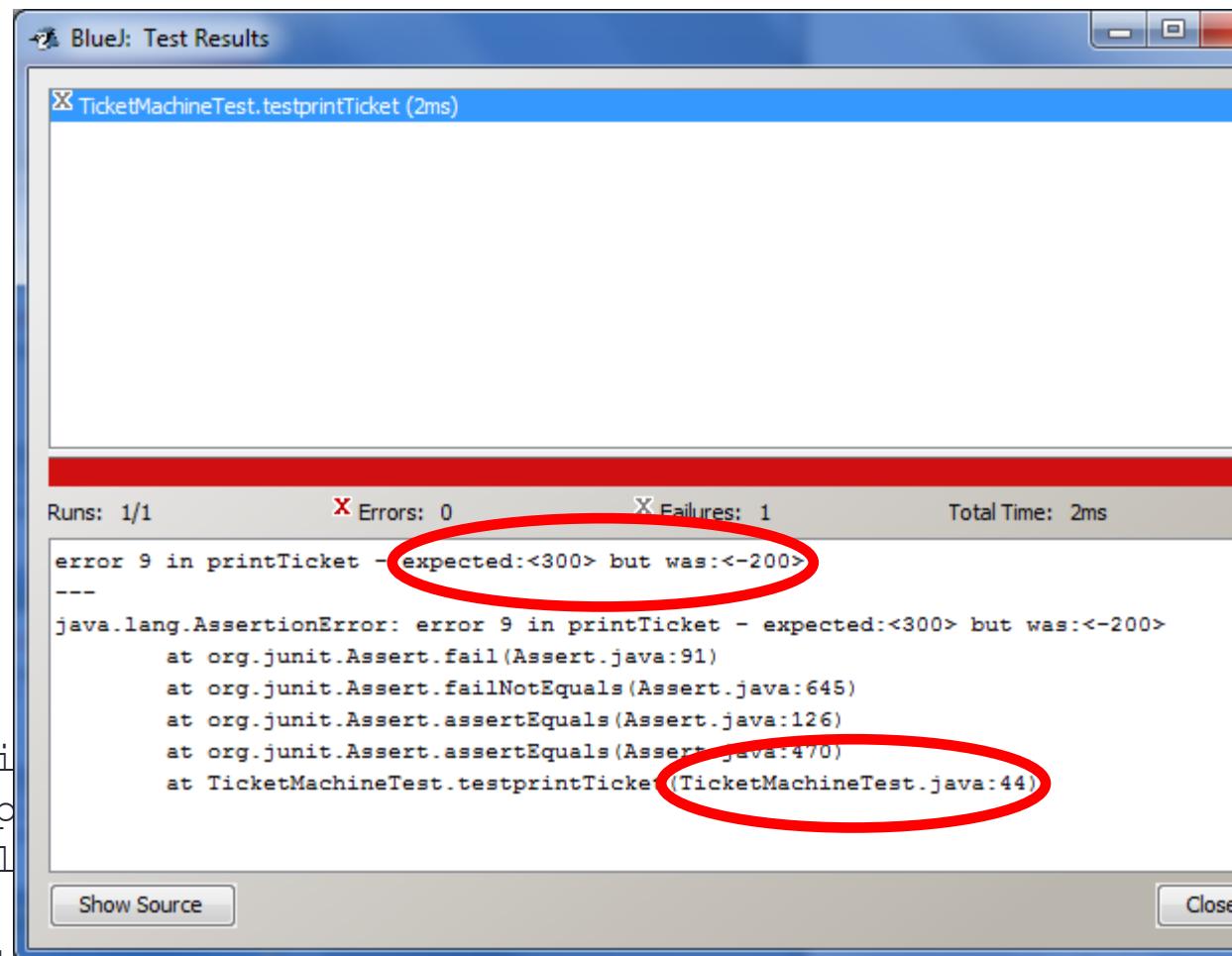
```
public void printTicket() {  
    // Simulate the printing of a ticket.  
    System.out.println("#####"); // printing elided  
  
    // Update the total collected with the price.  
    total += price;  
    // Update the balance.  
    balance -= price;  
}
```

# A third error!

- Now it is saying balance has gone negative

```
public void printTicket() {
    // Simulate the price being inserted.
    System.out.println("Inserting price " + price);

    // Update the total collected with the price.
    total += price;
    // Update the balance.
    balance -= price;
}
```



# Fix the third bug

- Transactions shouldn't be allowed if balance is too low

```
public void printTicket() {  
    // Simulate the printing of a ticket.  
    System.out.println("#####"); // printing elided  
  
    // Update the total collected with the price.  
    total += price;  
    // Update the balance.  
    balance -= price;  
}
```

# Fix the third bug

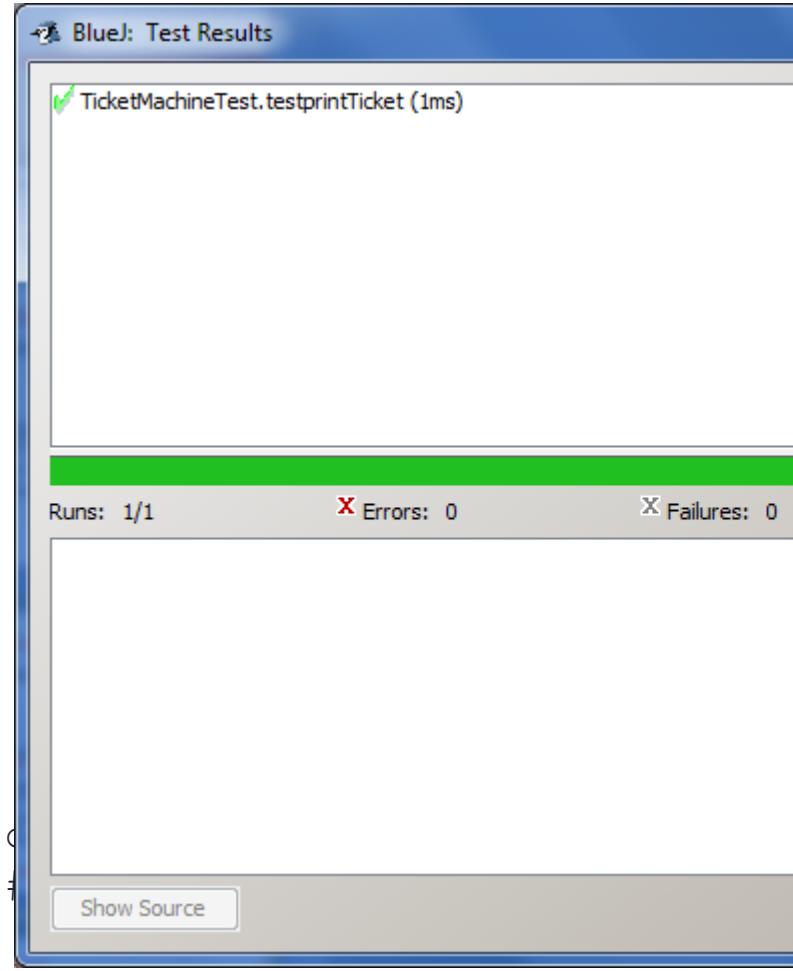
- Transactions shouldn't be allowed if balance is too low

```
public void printTicket() {  
    if(balance >= price)  
    {  
        // Simulate the printing of a ticket.  
        System.out.println("#####"); // printing elided  
  
        // Update the total collected with the price.  
        total += price;  
        // Update the balance.  
        balance -= price;  
    }  
    else // print an error msg  
}
```

# The tick is good!

- All tests passed

```
public void printTicket() {  
    if(balance >= price)  
    {  
        // Simulate the printing of a ticket.  
        System.out.println("#####  
        // Update the total collected with the price.  
        total += price;  
        // Update the balance.  
        balance -= price;  
    }  
    else // print an error msg  
}
```



# Limitations of JUnit

- Printing can't be tested
- JUnit can test only
  - Values returned by methods
  - Changes to an object's state
- JUnit can test only `public` methods
  - Sometimes methods which could/should be `private` are set `public` for testing
- JUnit can't see "inside" methods
  - That's what the debugger is for!

# A more subtle issue

- A subtle issue with relying on test cases to decide correctness is that it encourages you to write “the whole method”, then test it
- If a method is complex, a more fruitful approach may be to write “part of it”, then test, then write “more of it”, then test again, etc.
- So e.g. if a method returns an `ArrayList`, it might be better to first write a version that generates only the first item, then test that, then move on to the other parts of the method
- Or e.g. if a method sets several field variables, write a version that sets one, then test that, then the next one, then test again, etc.

# But passing all tests ≠ perfect program!

- Remember that even if/when your program passes all of the tests, that doesn't mean it's perfect
- The tests are normally a sample of the possible inputs that the program may need to process
  - Passing all tests means the program works for that sample, but doesn't guarantee that it works for other inputs
  - A later lecture in CITS1001 will discuss how to design good test data to maximise your confidence in your software
- **The correctness of your program is your responsibility**
- Testing (and other ways of demonstrating correctness) is a huge issue in software engineering
  - Much more in later units

# CITS1001

## 10. LIBRARIES AND TYPES

---

*Objects First With Java, Chapter 6*

# Lecture essentials

- Using libraries
  - Importing, Javadoc
- Math
  - static methods
- char
  - Unicode, casting, arithmetic
- String
  - Immutability, methods, ordering
- I/O
  - FileIO
- Class variables
  - final

# Libraries

- The libraries available in Java are accessible from the Java API

<https://docs.oracle.com/en/java/javase/11/docs/api/>

- There are many, many classes
  - Organised into *packages* of similar material
- Use built-in types and methods whenever they are suitable
  - If someone has already written the code that you need, using their class will usually be more efficient than writing your own
- We have already seen String, ArrayList, Random, etc.

# Importing classes

- Classes or packages can be imported explicitly, e.g.

```
import java.util.Random;  
import java.util.*;
```

- This enables you to use the short name in your code, e.g.

```
Random r = new Random();
```

- Some library classes are imported automatically
  - Including many that you may use frequently, e.g. Math, String, Integer, Character, Boolean, etc.

# Library documentation

- All library classes come with extensive documentation which tells you
  - What the class is called
  - How to create objects
  - What methods are available and what they do
- <https://docs.oracle.com/en/java/javase/11/docs/api/>
- This is everything you need as a user of the class
  - i.e. the external view of the class
- This documentation is derived from the Javadoc comments associated with each class, constructor, and method
  - To be discussed later in the unit

# Library documentation

- The documentation *does not* include any details on
  - Private fields (most fields are kept private)
  - Private methods
  - The bodies of any methods
  - i.e. the internal view of the class
- These details are part of the implementation of the class
  - They may be changed in the future, as long as the external view is not affected
  - As long as the external view of the class is unchanged, client code does not need to be changed

# An aside: public vs. private

- The access modifiers `public` and `private` control which fields, methods, and constructors in a class can be accessed/invoked from other classes
- `public type x1;`
  - `x1` can be accessed from any class in the program
- `public type m1 (...) { ... }`
  - `m1` can be invoked from any class in the program
- `private type x2;`
  - `x2` can be accessed only from the class where it is defined
- `private type m2 (...) { ... }`
  - `m2` can be invoked only from the class where it is defined
- More on this issue later in the unit

# Code completion

- The BlueJ editor supports lookup of methods
- Use *Ctrl-space* after a method-call dot to bring up a list of available methods
- Use the mouse and *Enter* to select a highlighted method
- Use code completion to help you remember method names in library classes

# Math

- Math defines pretty much all mathematical functionality that you will ever need
- Methods like abs, pow, max, min, sin, cos, tan, etc.
- Constants like PI and E
- You can use these with the *class name*, e.g.

```
int x      = -6;  
int y      = Math.abs(x);      // sets y to 6  
double z  = Math.pow(x, 3);   // sets z to -216.0  
z         = Math.PI * 2;      // sets z to 6.28...  
long w    = Math.round(z);   // sets w to 6
```

# Math

- Math defines pretty much all mathematical functionality that you will ever need
- Methods like `abs`, `pow`, `max`, `min`, `sin`, `cos`, `tan`, etc.
- Constants like PI and E
- You can use these with the *class name*, e.g.

```
int x      = -6;  
int y      = Math.abs(x);      // sets y to 6  
double z  = Math.pow(x, 3);   // sets z to -216.0  
z         = Math.PI * 2;      // sets z to 6.28...  
long w    = Math.round(z);    // sets w to 6
```

class name, not object name

# Static methods

- All of the methods in Math are static methods, e.g.

```
public static double pow(double a, double b)  
// returns a raised to the power b
```

- This is because the values they return don't depend on the state of an object, only on the arguments provided
- We can call a static method belonging to Math *without creating an object first*
  - In fact it is impossible to create an object belonging to Math
  - They are sometimes called *class* methods

# char

- `char` variables represent all types of characters
  - Letters (upper- and lower-case), digits, punctuation
  - Formatting characters, e.g. carriage-return '`\n`', tab '`\t`', etc.
  - Special characters, e.g. smiley, love heart, the Ace of Spades, etc.
  - Characters used in other languages, e.g. Arabic, Chinese, etc.
  - etc., etc., etc.
- In the (standard) Unicode system, every character is represented inside the machine by an integer
  - <http://www.tamasoft.co.jp/en/general-info/unicode-decimal.html>
- Characters are written in single quotes , e.g. '`A`', '`5`', '`;`'
  - They can also be written using their Unicode values in hexadecimal, e.g. '`\u0041`' is the same character as '`A`'

# Unicode

- Just a small part of the tables...
  - Note that these values are in decimal

## Unicode Table (Decimal)

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
0	Γ	γ	Ľ	Ľ		-	•	♦						়	়	+	◀	↕	॥	
20	¶	⊥	⊤	†	↑	†	→	←	Ľ	↔	▲	▼	!	"	#	\$	%	&	'	
40	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	
60	<	=	>	?	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~		€	,	f	„	…	†	‡	^	%	Š	č	
140	Œ	Ž												TM	š	>	œ		ž	Ŷ
160	í	ć	£	¤	⌘		§	~	©	¤	«	¬		®	—	°	±	²	³	
180	‘	μ	¶	·	,	¹	º	»	¼	½	¾	¿	À	Á	Â	Ã	Ä	Å	Æ	Ç
200	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û
220	Ü	Ý	Þ	ß	à	á	â	ã	å	ä	æ	ç	è	é	ê	ë	í	í	í	
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	Ā	ā	Ā	
260	Ą	ą	Ć	ć	Ĉ	ĉ	Ć	ć	Č	č	Đ	đ	Đ	đ	Ē	ē	Ē	ě	Ē	
280	Ę	ę	Ě	ě	Ĝ	ĝ	Ĝ	ĝ	Ĝ	ĝ	Ĝ	ĝ	Ĥ	ĥ	Ĥ	ĥ	Ĭ	ĭ	Ĭ	
300	Ĩ	ĩ	Ļ	ļ	Ī	ī	Ĳ	ij	Ĵ	ȷ	Ķ	ķ	Ķ	ķ	Ĺ	í	Ĺ	ŕ	Ĺ	
320	Ľ	ľ	Ľ	Ľ	Ń	ń	Ń	ń	Ń	ń	Ń	ń	Ń	ń	Ő	ő	Ő	ő	Œ	œ
340	Ŕ	ŕ	Ŗ	ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	Ŗ	
360	Ũ	ũ	Ũ	ũ	Ũ	ũ	Ũ	ũ	Ũ	ũ	Ũ	ũ	Ũ	ũ	Ŵ	ŵ	Ŷ	ŷ	Ŷ	
380	ż	Ž	ž	ſ	ƀ	Ɓ	Ɓ	ƀ	ƀ	ƀ	Ծ	Ծ	Ծ	Ծ	Ծ	Ծ	Ճ	Ճ	Ճ	
400	Ը	Ժ	ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	Ժ	
420	Պ	պ	Ր	ր	Ր	ր	Ր	ր	Ր	ր	Ր	ր	Ր	ր	Ո	օ	Օ	օ	Ը	
440	Ծ	ծ	Զ	զ	Յ	յ	Յ	յ	Յ	յ	Յ	յ	Յ	յ	Յ	յ	Յ	յ	Յ	
460	nj	Ă	ă	Ĭ	î	Ӧ	ӧ	Ӧ	Ӧ	Ӧ	Ӧ	Ӧ	Ӧ	Ӧ	Ӧ	Ӧ	Ӧ	Ӧ	Ӧ	
480	Ā	ā	Ā	ā	Ē	ē	Ē	ē	Ē	ē	Ē	ē	Ē	ē	Ӄ	Ӄ	Ӄ	Ӄ	Ӄ	

# Casting

- *Casting* means to change a value from one type to a “corresponding” value in another type
- We can cast `char` values to their Unicode `int` values and vice versa, e.g. (try these in the Code Pad)

(char) 65 → 'A'

(int) 'B' → 66

- Casting can also be used in other contexts, e.g.

(double) 65 → 65.0

(int) 5.2 → 5

**note the truncation**

# char arithmetic

- You do ***not*** need to memorise any of the values from the Unicode tables
- However it is useful to be aware of certain patterns in the table, e.g.
  - The codes for 'a' ... 'z' are consecutive
  - The codes for 'A' ... 'Z' are consecutive
  - The codes for '0' ... '9' are consecutive
- This allows for arithmetic over `char`, using casting between `char` and `int`

# char arithmetic

- You do ***not*** need to memorise every character from the Unicode tables
- However it is useful to be familiar with the table, e.g.
  - The codes for 'a' ... 'z' ↪
  - The codes for 'A' ... 'Z' ↪
  - The codes for '0' ... '9' ↪
- This allows for arithmetic between `char` and `int`

Unicode Table (Decimal)

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
0	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
20	ؔ	؄	؅	؆	؇	؈	؉	؊	؋	،	؍	؎	؏	ؐ	ؑ	ؔ	ؔ	ؔ	ؔ	
40	؂	؃	؄	؅	؆	؇	؈	؉	؊	؋	،	؍	؎	؏	ؐ	ؑ	ؔ	ؔ	ؔ	
60	؂	؃	؄	؅	؆	؇	؈	؉	؊	؋	،	؍	؎	؏	ؐ	ؑ	ؔ	ؔ	ؔ	
80	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
100	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
120	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
140	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
160	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
180	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
200	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
220	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
240	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
260	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
280	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
300	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
320	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
340	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
360	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
380	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
400	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
420	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
440	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
460	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	
480	ؐ	ؑ	ؒ	ؔ	ؕ	ؖ	ؗ	ؘ	ؙ	ؚ	؛	؜	؝	؞	؟	ؠ	ء	آ	أ	

# char arithmetic examples

```
char c = 'J';  
c += 1; // sets c to 'K'  
c -= 9; // sets c to 'B'  
char d = 'H';  
boolean b = c < d; // sets b to true  
b = c >= 'C'; // sets b to false  
for (char ch = 'a'; ch <= 'z'; ch++)  
    {} // one iteration for each lower-case letter  
c *= 2; // legal code, but don't ever do this!
```

- Be clear on the difference between e.g.
  - 'c', which is a `char` literal
  - "c", which is a `String` literal of length 1
  - c, which is a variable name (could be of any type)

# String

- String that we have seen already is an object type which represents a sequence of characters
- Strings are written in double quotes
  - e.g. "CITS1001" is a sequence of length 8
  - e.g. "A" is a sequence of length 1
  - "" is a sequence of length 0
  - " " is a sequence of length 1 containing a space
  - " " is a sequence of length 2 containing two spaces
- Be clear on the difference between e.g.
  - 8, which is an int
  - '8', which is a char
  - "8", which is a String of length 1

# String

- String is a class defined in the library
- Strings in Java are immutable objects
  - They cannot be changed after they are created
  - If you want a different string, you must create a new one
  - Contrast this with e.g. TicketMachine
- The String library contains a large number of useful methods, e.g.

```
String s1 = "R2D2";  
String s2 = s1.toLowerCase(); // sets s2 to "r2d2"
```

- Be clear that this returns a new String
  - s1 is unchanged

# Intermission

```
if(condition)
```

---

```
if(condition == true)
```

---

```
if(String.valueOf
```

```
(condition).equals("true"))
```



# Methods in the String class

- Two basic and crucial methods are

```
public int length()
```

- Returns the number of characters in the string

```
public char charAt(int index)
```

- Returns the character at the given index, where indexing starts at 0

```
String s = "Will.i.am";
```

```
int x      = s.length(); // sets x to 9
```

```
char c      = s.charAt(8); // sets c to 'm'
```

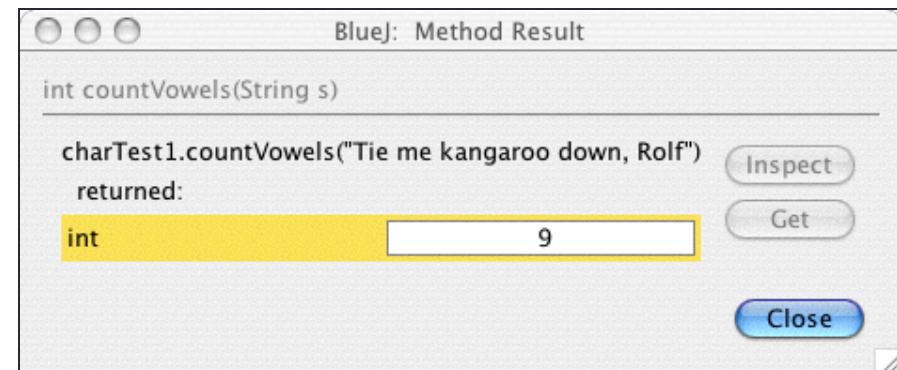
# Processing a String

- These two methods give us the fundamental mechanism for inspecting each character of a `String` in turn

```
public void inspectString(String s) {  
    for (int i = 0; i < s.length(); i++)  
    {  
        char ch = s.charAt(i);  
        // Do something with ch  
    }  
}
```

# Counting vowels

```
public int countVowels(String s) {
    int numVowels = 0;
    for (int i = 0; i < s.length(); i++) {
        char ch = s.charAt(i);
        if (ch == 'a' || ch == 'A')
            numVowels++;
        if (ch == 'e' || ch == 'E')
            numVowels++;
        if (ch == 'i' || ch == 'I')
            numVowels++;
        if (ch == 'o' || ch == 'O')
            numVowels++;
        if (ch == 'u' || ch == 'U')
            numVowels++;
    }
    return numVowels;
}
```



# Ordering strings

- Strings can be compared for ordering purposes
  - The standard ordering is *lexicographic ordering*
  - The same as in a standard English dictionary
- The following words are lexicographically ordered
  - aardvark, app, application, ban, band, cure
- How are two words ordered?
- By the first letter in which they differ, if any
  - So **ban** comes before **cure**, and
  - **aardvark** comes before **apple**
- Otherwise by their length
  - So **ban** comes before **band**, and
  - **app** comes before **application**

# Ordering strings

- This is achieved with the library method `compareTo`

```
public int compareTo(String anotherString)
```

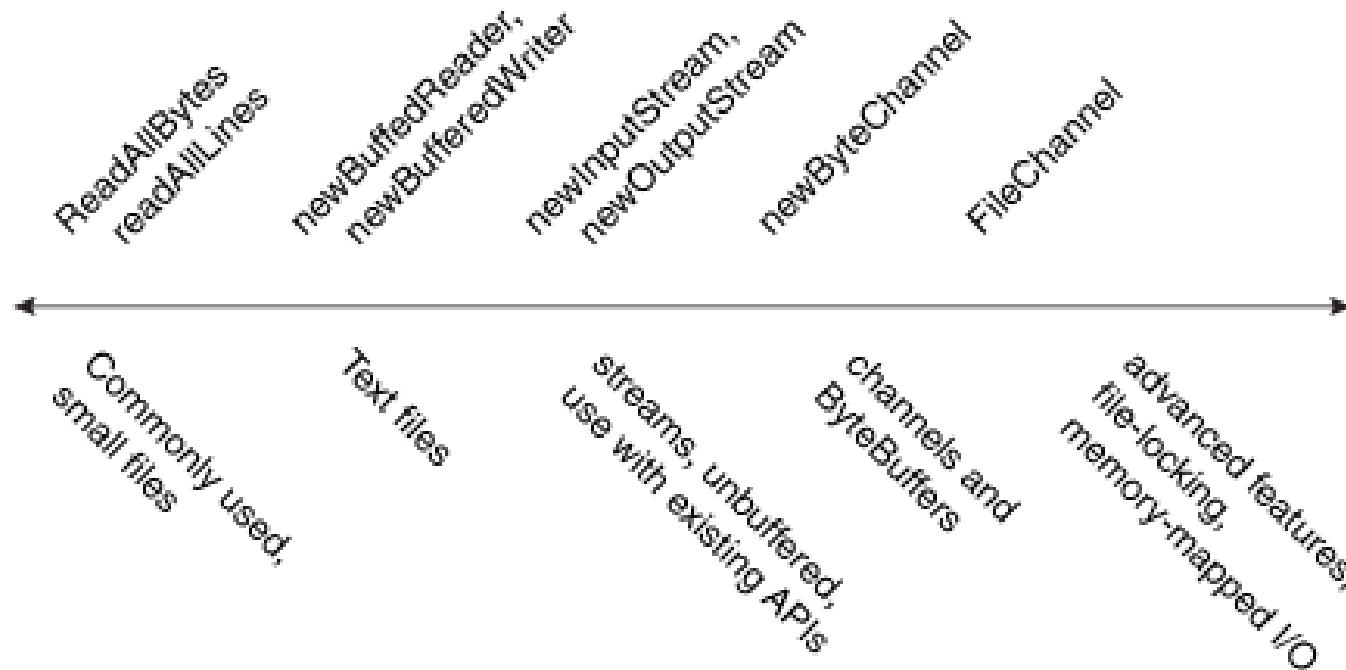
- In `s1.compareTo(s2)`
  - `s1` is called the **target String**
  - `s2` is called the **argument String**
- `s1.compareTo(s2)` returns
  - A negative number if the target comes *before* the argument,  
e.g. `s1 = "b"`, `s2 = "c"`
  - A positive number if the target comes *after* the argument,  
e.g. `s1 = "b"`, `s2 = "a"`
  - Zero if the target is equal to the argument,  
e.g. `s1 = s2 = "b"`

# Ordering strings

- But be careful!
- Technically, ordering is based on Unicode values
- So e.g. “Xylophone” comes before “apple”
  - Because the Unicode value for ‘x’ is less than that for ‘a’
- It is safe to rely on this ordering only if words have the same case (lower or upper)
- Be careful too when comparing String numbers
  - e.g. “452” comes before “612”, which seems intuitive
  - But “452” also comes before “61” and “6”
- These comparisons are never safe for punctuation, symbols, etc.

# Files

- As in most languages, File I/O in Java is quite complicated
  - File size? Formats? Buffering? Streaming?
- There is an excellent tutorial at
  - <http://docs.oracle.com/javase/tutorial/essential/io/file.html>



# In CITS1001, FileIO will suffice

```
FileIO fio = new FileIO("Test.txt");
```

- Creates a FileIO object with two fields
  - String fio.filename will contain "Test.txt"
  - ArrayList<String> fio.lines will contain <"abc", "De f?", "", "12 34 56">
- These variables can be read via accessor methods
  - Check out the code for their names
- Be wary of different operating systems, blank lines, and trailing carriage-returns!

The diagram illustrates the state of a FileIO object. It shows a yellow rectangular box representing the object's state. At the top, it contains the string "Test.txt". Below this is a horizontal dashed line. Underneath the line, the object contains four lines of text: "abc", "De f?", "", and "12 34 56".

# Class methods

- We have already seen that *class methods* are indicated by the keyword `static` in their header
  - e.g. the methods in the `Math` library
- A method can be `static` only if its behaviour depends only on the arguments to the method
  - The method thus does the same thing for all possible objects
- The method effectively belongs to the class itself
  - It can be invoked with the class name, rather than an object name

```
public static double pow(double a, double b)  
// returns a raised to the power b
```

```
double z = Math.pow(2,10); // sets z to 1024.0
```

# Class variables

- Variables sometimes also should be viewed as belonging to the class in which they are defined, rather than objects belonging to that class
  - These are known as *class variables*
- When would this be appropriate?
  - When the variable should **always** have the same value for all possible objects

# Example

- Imagine a class BouncingBall
- A ball would need (at least) three fields

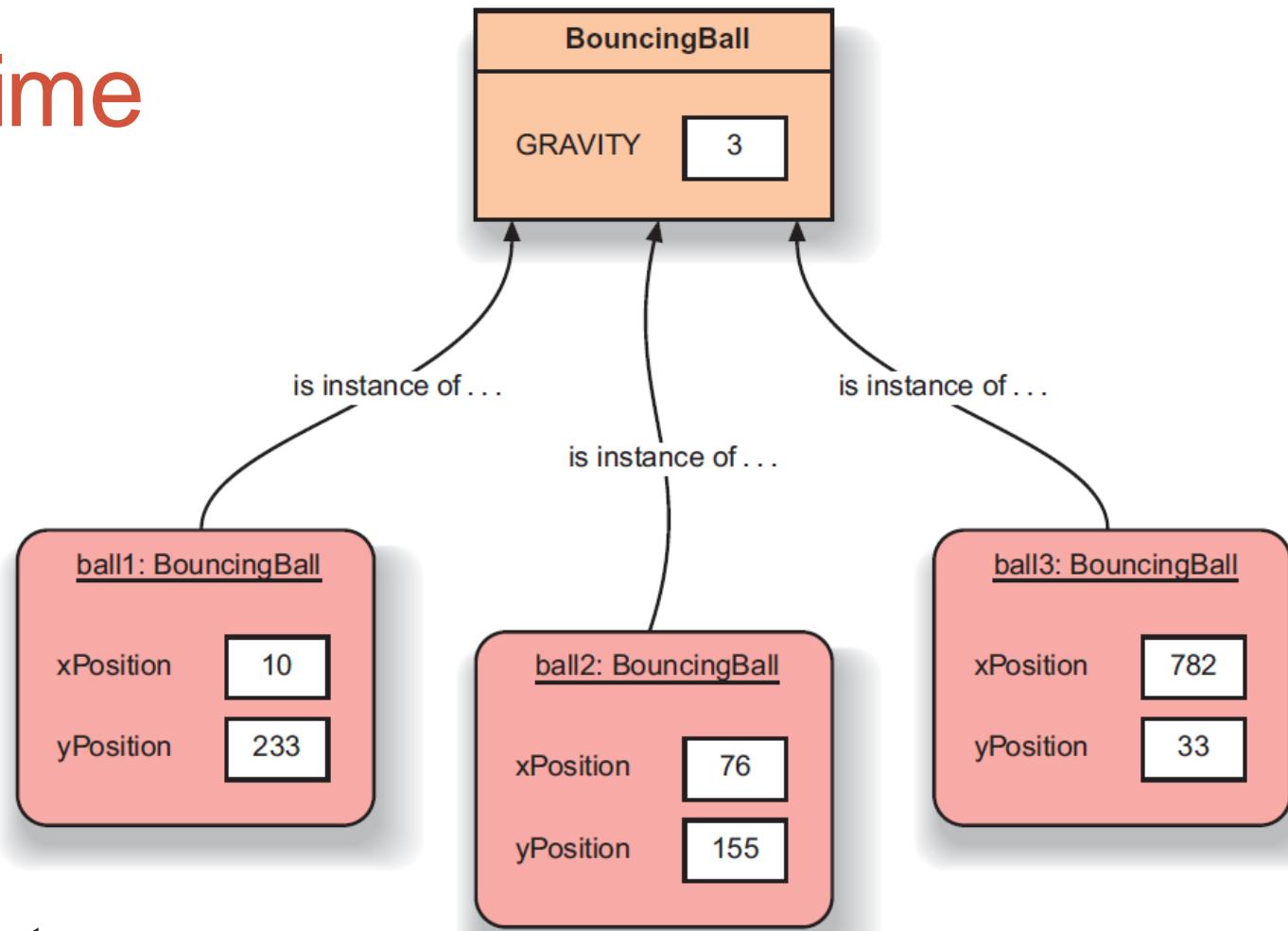
```
class BouncingBall {  
    private int GRAVITY;      // physics of the world  
    private int xPosition;   // where the ball is  
    private int yPosition;  
  
    ...  
}
```

# Example

- But this set-up would allow different BouncingBall objects to have different gravitational constants
  - Three objects would have three different GRAVITY variables
- Instead what we want is for the GRAVITY field to belong to the class, rather than the objects themselves
  - Thus there will be only one GRAVITY variable to which all objects refer
- This is (again) achieved with the keyword static

```
class BouncingBall {  
    private static int GRAVITY; // physics of the world  
    private int xPosition;      // where the ball is  
    private int yPosition;  
  
    ...  
}
```

# At run-time



- Three objects
  - Three copies each of `xPosition` and `yPosition`
  - Only one copy of `GRAVITY`

# Other examples

- Other physical constants, e.g. boiling point of water
  - Mathematical constants, e.g. `Math.PI` and `Math.E`
  - Bank accounts sharing an interest rate
  - Windows that must have the same dimensions
- 
- But why are `GRAVITY`, `PI`, and `E` written in ALL CAPS?
  - As well as being class variables shared between all objects belonging to the relevant class, they also need to be constant
    - i.e. once initialised, their value never changes

# final variables

- This gives the actual code for BouncingBall

```
class BouncingBall {  
    private static final int GRAVITY = 3;  
                           // physics of the world  
    private int xPosition; // where the ball is  
    private int yPosition;  
  
    ...  
}
```

- GRAVITY is a class variable that is also `final`
  - `final` variables must be initialised immediately and can never be changed

# CITS1001

## 11. ARRAYS – PART 1

---

*Objects First With Java,  
Chapter 7*

# Lecture essentials

- Arrays
- Declaring and constructing arrays
- Using and returning arrays
- Aliasing
- Arrays of objects
- 2D arrays

# Fixed-size collections

- An `ArrayList` is used when the size of a collection is not known in advance, or might vary
- But in some situations, the collection size can be pre-determined from the data
- For this situation, a special fixed-size collection type is available: the `array`
  
- Arrays can store object references or primitive values
- Arrays use a special more-concise syntax
- Arrays are very common in a wide range of programming languages

# Uses of arrays

- Arrays are used when we have large numbers of same-typed values or objects that we want to operate on as a collection
  - A collection of temperatures that we want to average
  - A collection of student marks that we want to analyse
  - A collection of names that we want to sort
- e.g. Bureau of Meteorology monthly data
  - We know there are twelve months in a year

## ALBANY

Max	25.1	25.1	24.1	21.5	18.7	16.6	15.7	15.9	17.4	18.8	20.8	23.4
Min	13.5	14.3	13.3	11.6	9.8	8.1	7.4	7.4	7.9	9.0	10.6	12.3
Rain	28	25	29	66	102	104	126	104	81	80	46	24

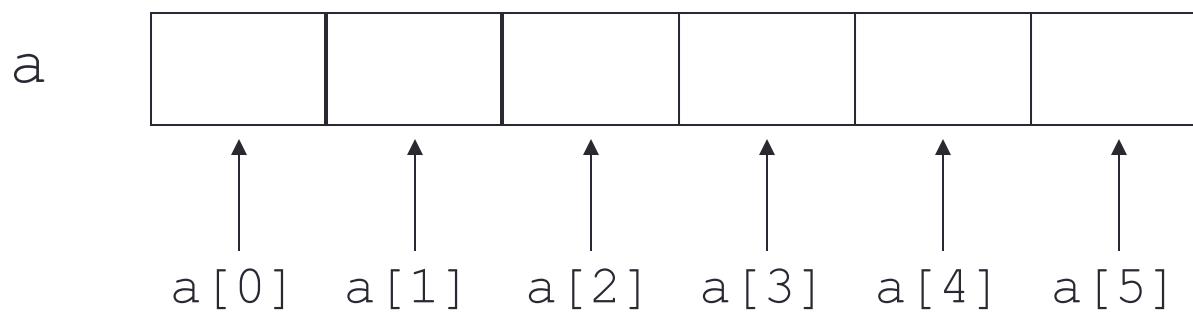
## PERTH AIRPORT

Max	31.4	31.7	29.5	25.2	21.4	18.7	17.6	18.3	20.0	22.3	25.4	28.5
Min	16.7	17.4	15.7	12.7	10.2	9.0	8.0	7.9	8.8	10.1	12.4	14.6
Rain	8	14	15	46	108	175	164	117	68	48	25	12



# Arrays

- An array is an indexed sequence of variables of the same type



- The array is called `a`
- Its elements are called `a[0]`, `a[1]`, `a[2]`, etc.
  - Each element is a separate variable
- Notice that indexing starts from 0
  - Same as with `ArrayList` and `String`

# Declaring arrays

- An array variable is declared using the usual syntax
  - The [ ] denotes an array variable

```
int[] a;
```

- Declares `a` to be a variable representing an array of `ints`

```
double[] temps;
```

- Declares `temps` to be a variable representing an array of `doubles`

```
String[] names;
```

- Declares `names` to be a variable representing an array of `Strings`

```
Student[] marks;
```

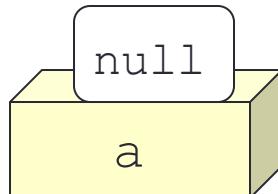
- Declares `marks` to be a variable representing an array of `Students`

# Creating Arrays I

- An array is an *object* in a Java program
- Therefore the declaration simply creates a variable to “point to” the array, but does not create the array itself
- Hence the declaration

```
int[] a;
```

allocates a space called `a`, big enough to hold an *object reference*, and initialised to `null`

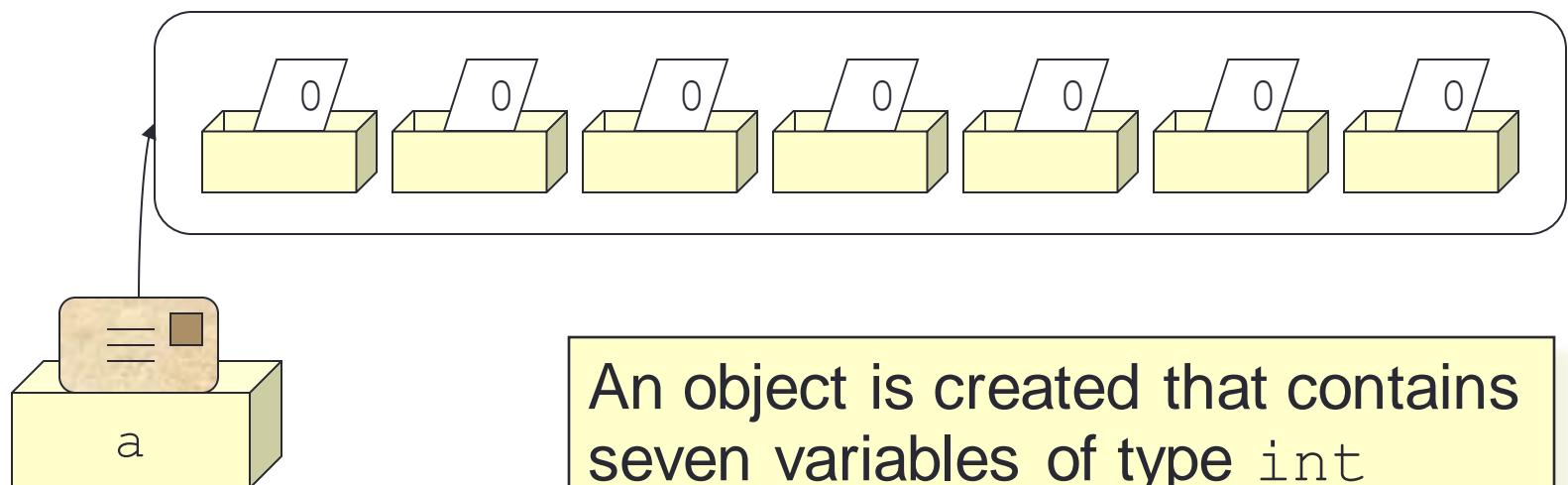


The declaration allocates space for the array reference, but **not** for the array itself

# Creating Arrays II

- In order to actually create the array, we must use the keyword `new` (just like creating any other object)

```
a = new int[7];
```



# The size of an array

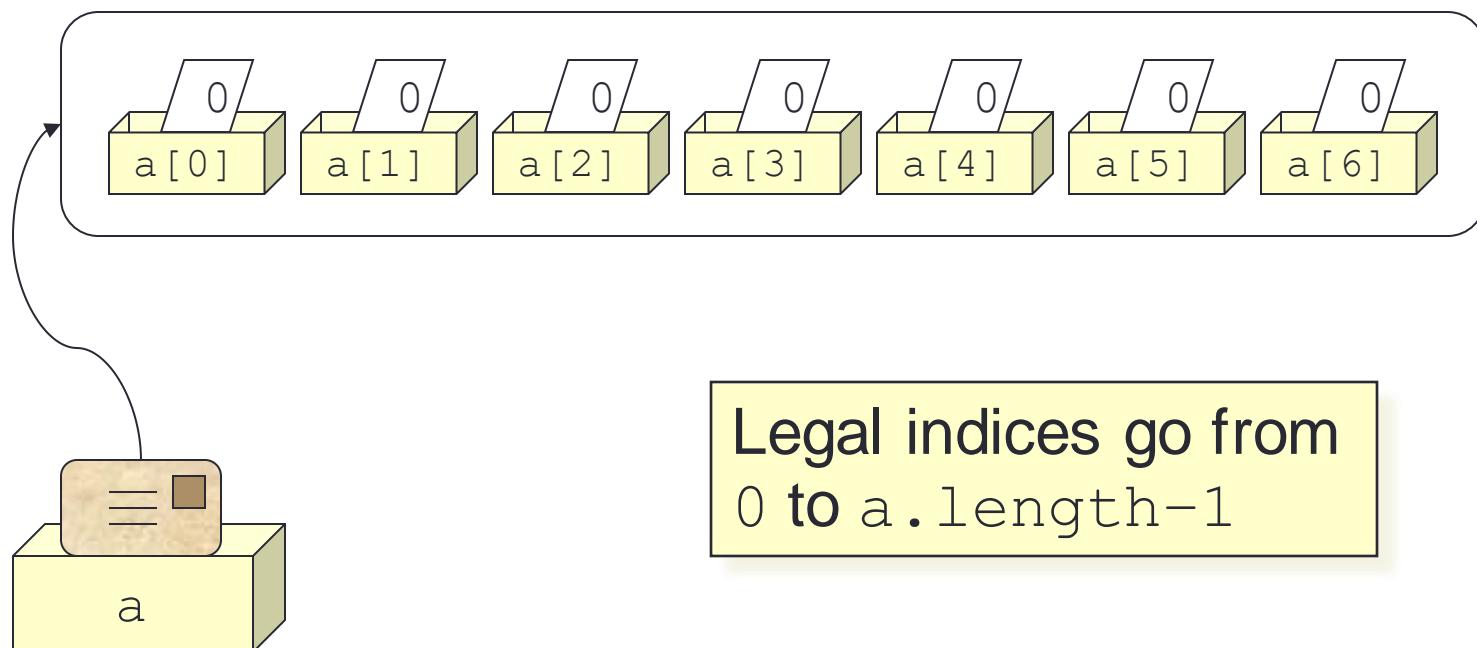
- The expression `a.length` denotes the size of the array currently pointed to by the variable `a`
  - We will see examples of its use later
- The size of an array is a property of the object that is created, not the variable that points to it
- An array variable can point to different arrays at different times, possibly with different sizes

```
int[] a;  
a = new int[7];  
System.out.println(a.length);  
a = new int[666];  
System.out.println(a.length);
```

7  
666

# Creating Arrays III

- The seven variables do not have individual names
- They are referred to by the array name, and their index



# Referencing array elements

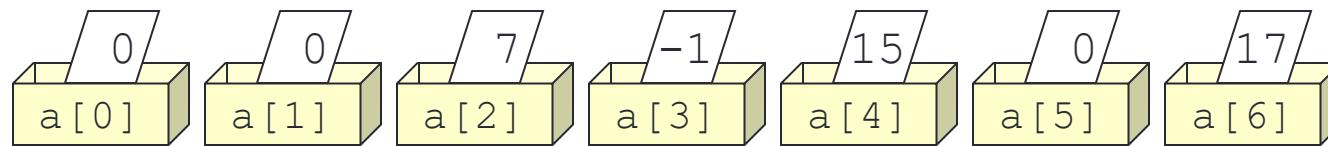
- Array elements can be used in the same ways and in the same contexts as any other variable of that type

```
a[4] = 15;
```

```
a[2] = 7;
```

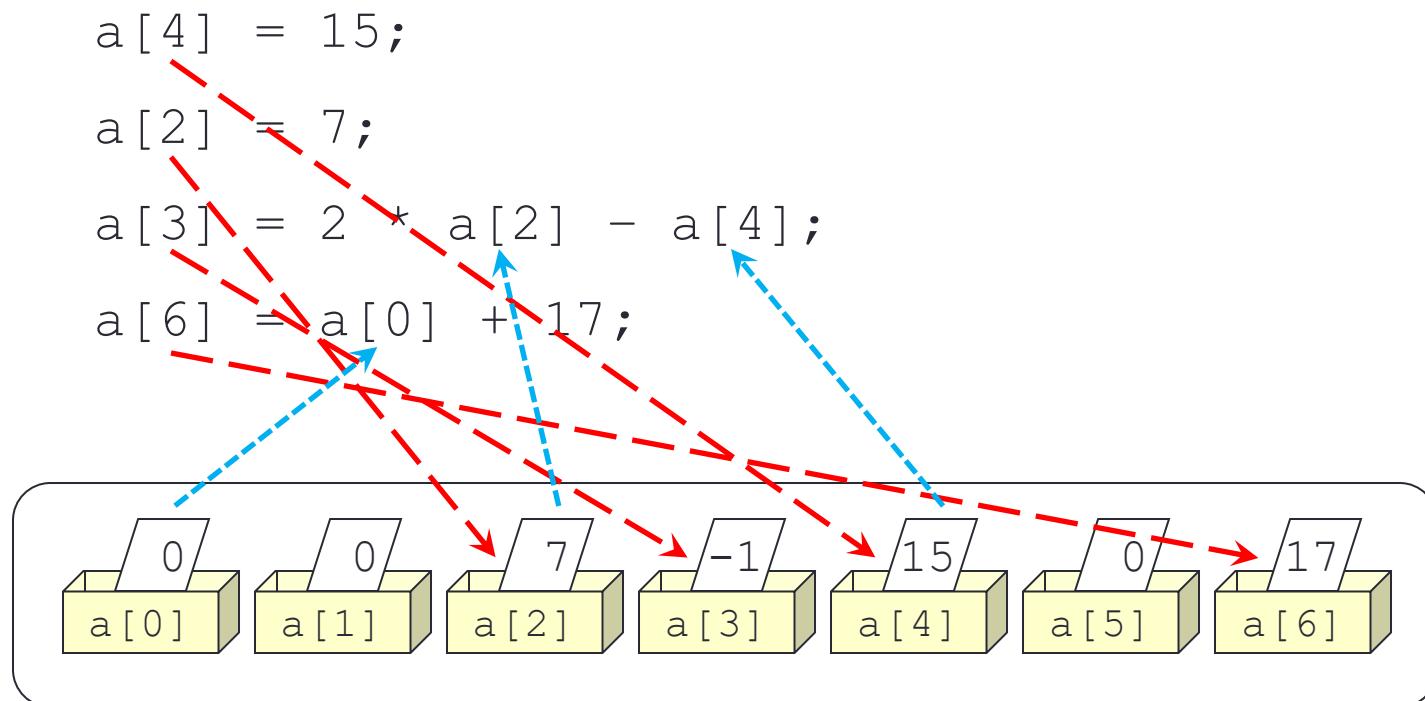
```
a[3] = 2 * a[2] - a[4];
```

```
a[6] = a[0] + 17;
```



# Referencing array elements

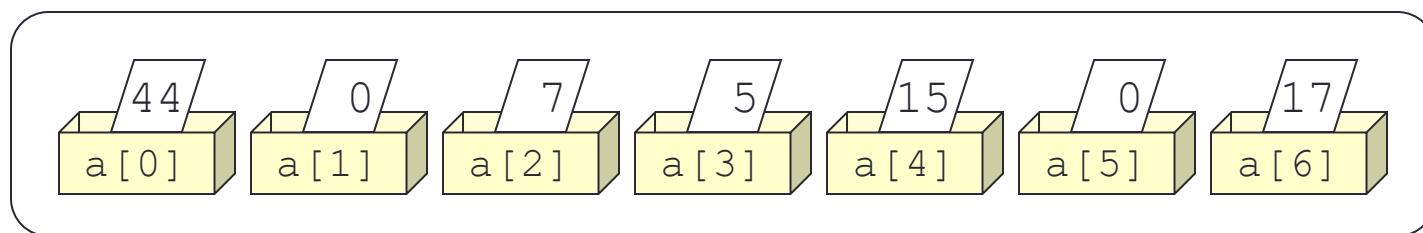
- Array elements can be used in the same ways and in the same contexts as any other variable of that type



# Indexing arrays

- A lot of the power of arrays comes from the fact that the index can be a *variable* or an *expression*

```
int x = 3;  
a[x] = 5;  
a[3-x] = a[2*x] * 2 + 10;
```

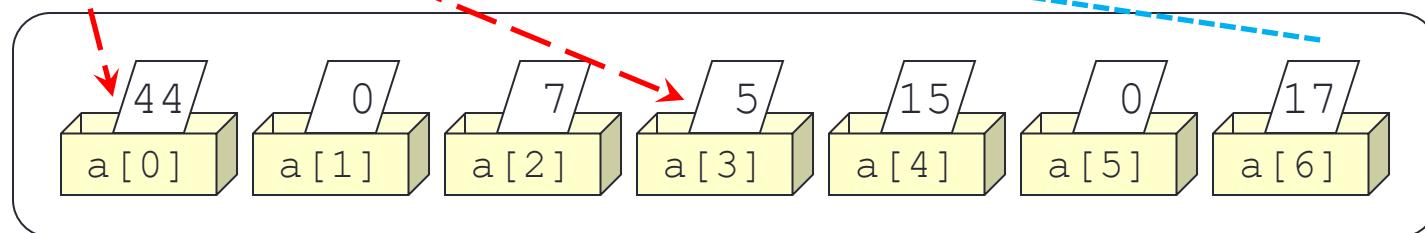


- This is especially useful when arrays are manipulated inside loops

# Indexing arrays

- A lot of the power of arrays comes from the fact that the index can be a *variable* or an *expression*

```
int x = 3;  
a[x] = 5;  
a[3-x] = a[2*x] * 2 + 10;
```



- This is especially useful when arrays are manipulated inside loops

# Summing the integers in an array

- `sum( { 5, 8, 6, 9, 7 } )` returns 35

```
private int sum(int[] a)
{
    int sum = 0;
    for (int i : a)
    {
        sum += i;
    }
    return sum;
}
```

accumulating variable

- Here `i` is an element of the array `a`
  - For-each loops work the same way with arrays as with `ArrayLists`

# Or using a for loop

```
private int sum(int[] a)
{
    int sum = 0;
    for (int i = 0; i < a.length; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

- Here `i` is being used as an index into the array `a`
- The use of `a.length` to stop the loop means that it will iterate for all elements in `a`

# Finding the biggest integer in an array

- `max( { 5, 8, 6, 9, 7 } )` returns 9

```
private int max(int[] a)
{
    int max = a[0];
    for (int i : a)
    {
        if (i > max) max = i;
    }
    return max;
}
```

accumulating variable

- Note that this is only well-defined for non-empty arrays
- Again `i` is an element of the array `a`

# Or using a for loop

```
private int max(int[] a)
{
    int max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (a[i] > max) max = a[i];
    }
    return max;
}
```

- Again `i` is being used as an index into the array `a`

# Finding the index of the biggest integer

- `indexOfMax({5, 8, 6, 9, 7})` returns 3

```
private int indexOfMax(int[] a)
{
    int k = 0;
    for (int i = 1; i < a.length; i++)
    {
        if (a[i] > a[k]) k = i;
    }
    return k;
}
```

The diagram shows three arrows originating from the variable 'k' in the code. One arrow points from the initial assignment `k = 0;` to the text 'accumulating variable'. Another arrow points from the update statement `k = i;` to the same text. A third arrow points from the final return statement `return k;` to the same text.

accumulating variable

- A for-each loop is awkward for this example



int



Integer

# Array literals

- An array literal is written using {}

```
int[] numbers = {3, 15, 4, 5};
```

- Array literals in this form can only be used in declarations
- Related uses require new

declaration,  
creation, and  
initialisation

```
int[] numbers;  
...  
numbers = new int[] {3, 15, 4, 5};
```

# Returning an array

- Suppose we want to return the running sums of `a`
  - e.g. `sums({2, 5, -8, 3})` should return `{2, 7, -1, 2}`
  - The  $i^{\text{th}}$  element in the result is the sum of the first  $i+1$  elements

```
private int[] sums(int[] a)
{
    int[] sums = new int[a.length];
    sums[0] = a[0];
    for (int i = 1; i < a.length; i++)
    {
        sums[i] = sums[i-1] + a[i];
    }
    return sums;
}
```

**create the result array**

**set up the values**

**return the result**

# Returning an array

- Suppose we want to average each pair of elements in a
  - e.g. `avs({2, 6, -8, 2})` should return `{4, -3}`

```
private int[] avs(int[] a)
{
    int[] avs = new int[a.length / 2];
    for (int i = 0; i < a.length - 1; i = i+2)
    {
        avs[i/2] = (a[i] + a[i+1]) / 2;
    }
    return avs;
}
```

**create the result array  
of the right length**

**return the result**

**set up the values**

# Returning an array

- An alternative formulation that loops over the elements of the result array, rather than the elements of  $a$

```
private int[] avs(int[] a)
{
    int[] avs = new int[a.length / 2];
    for (int i = 0; i < avs.length; i++)
    {
        avs[i] = (a[2*i] + a[2*i+1]) / 2;
    }
    return avs;
}
```

# Updating an array

- Sometimes instead of creating a new array, we want to update the old one
  - Returning sums again

```
private void sums(int[] a)
{
    for (int i = 1; i < a.length; i++)
    {
        a[i] = a[i-1] + a[i];
    }
}
```

- But this method doesn't return anything...
  - Before: a = {2, 5, -8, 3}

# Updating an array

- Sometimes instead of creating a new array, we want to update the old one
  - Returning sums again

```
private void sums(int[] a)
{
    for (int i = 1; i < a.length; i++)
    {
        a[i] = a[i-1] + a[i];
    }
}
```

- But this method doesn't return anything...
  - After: a = {2, 7, -1, 2} }

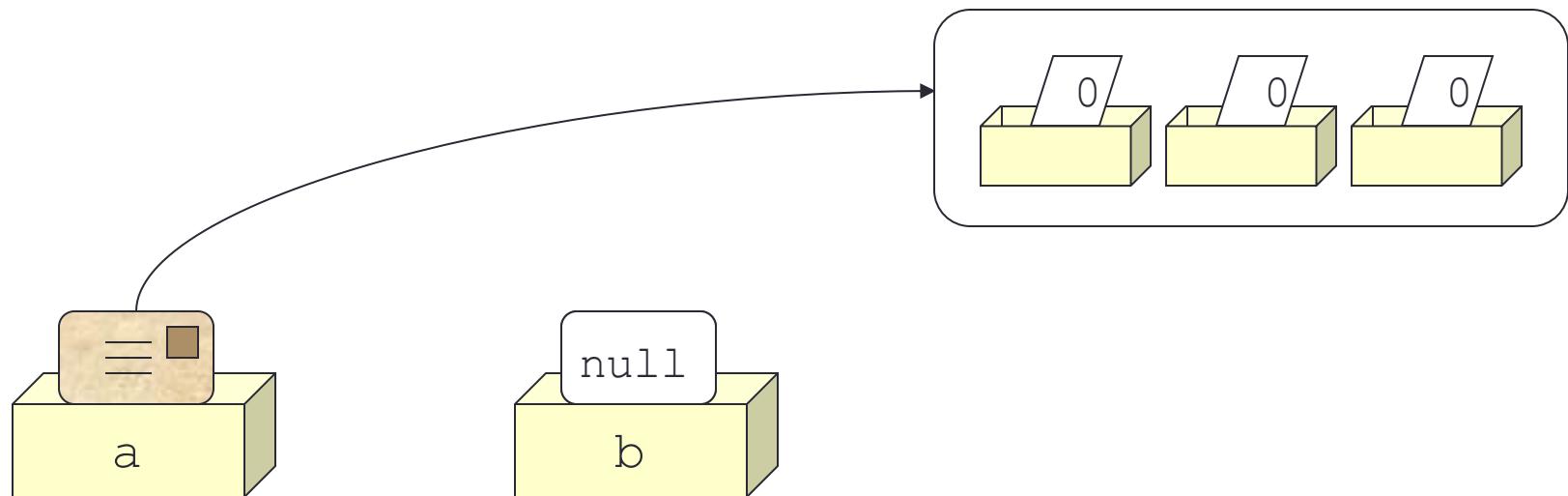
# An aside: arrays can share memory

```
int[] a, b;
```



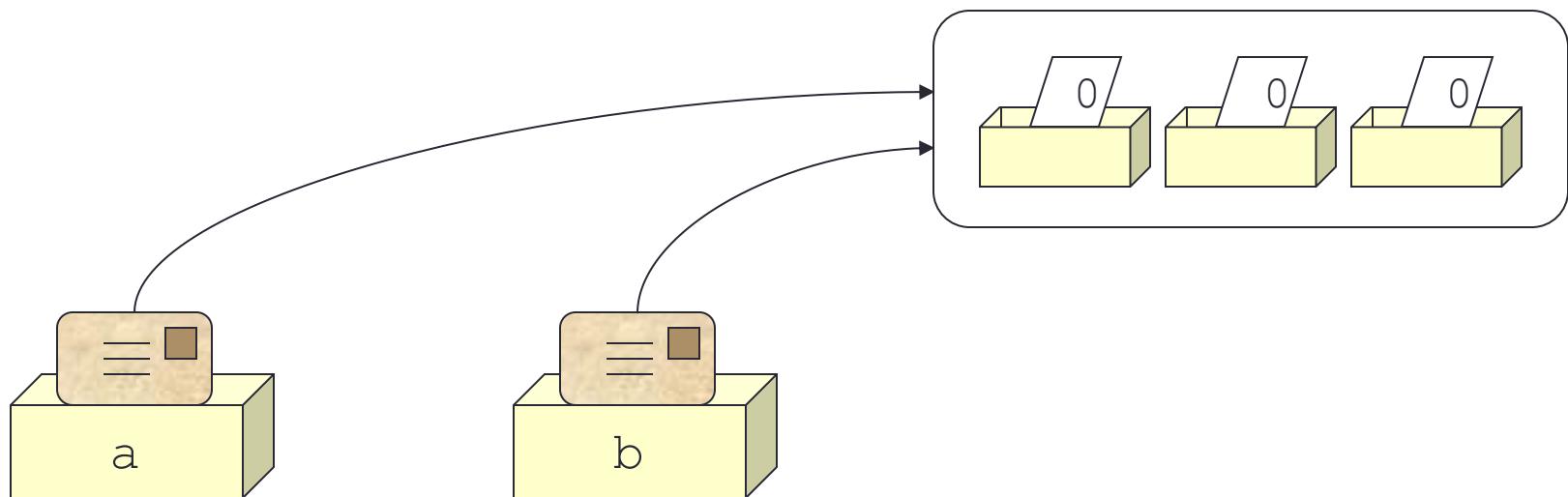
# An aside: arrays can share memory

```
int[] a, b;  
a = new int[3];
```



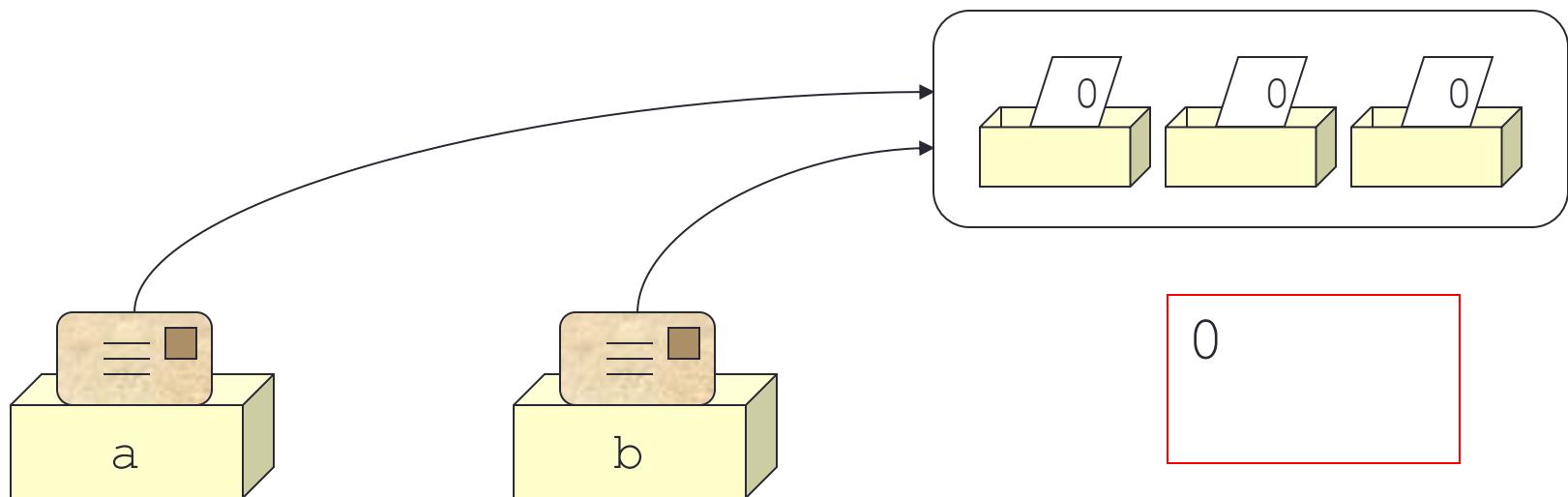
# An aside: arrays can share memory

```
int[] a, b;  
a = new int[3];  
b = a;
```



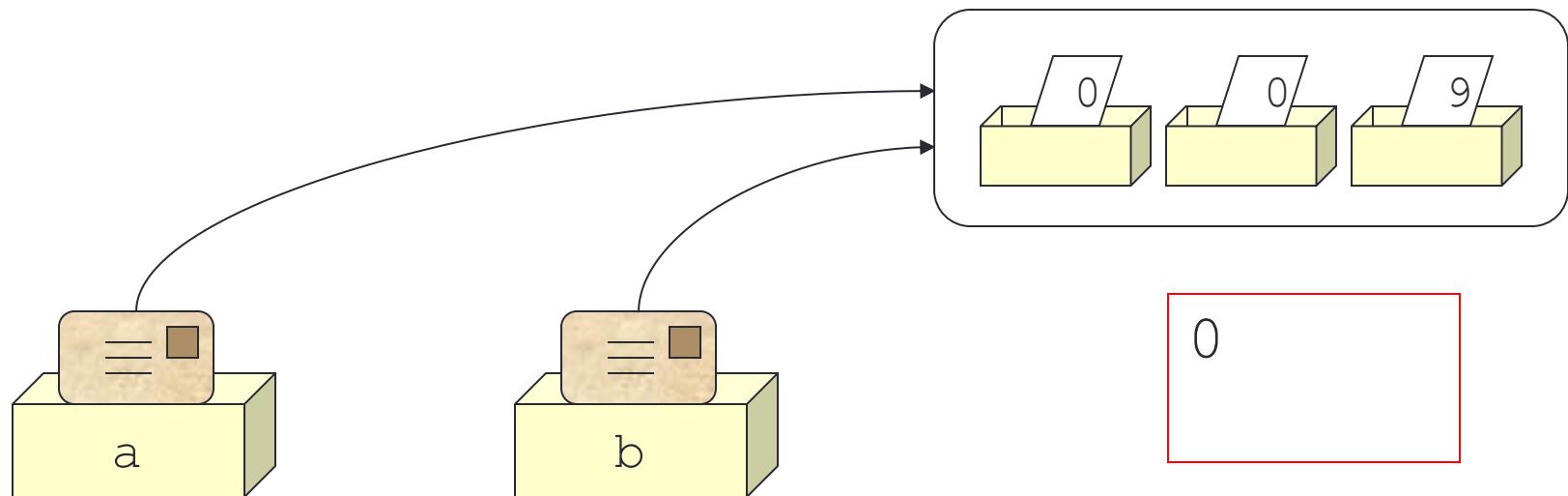
# An aside: arrays can share memory

```
int[] a, b;  
a = new int[3];  
b = a;  
System.out.println(b[2]);
```



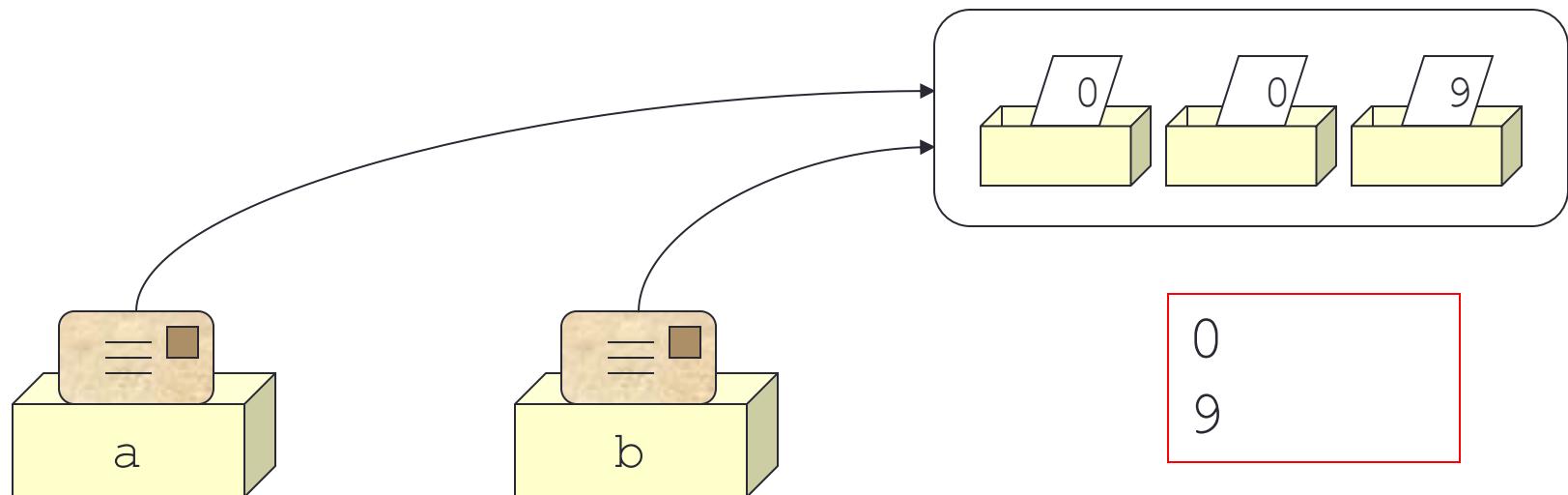
# An aside: arrays can share memory

```
int[] a, b;  
a = new int[3];  
b = a;  
System.out.println(b[2]);  
a[2] = 9;
```



# An aside: arrays can share memory

```
int[] a, b;  
a = new int[3];  
b = a;  
System.out.println(b[2]);  
a[2] = 9;  
System.out.println(b[2]);
```



# Aliasing

- This is called *aliasing*
- Basically one array with two names
- It applies in the same way to any other object-type too
- In some (advanced) applications aliasing might be used deliberately

To be continued in Part 2

# CITS1001

## 11. ARRAYS – PART 2

---

*Objects First With Java,  
Chapter 7*

# Lecture essentials

- Arrays of objects
- 2D arrays

# Changes to parameters are usually lost

- When we call a method with a parameter of a primitive type, any updates to that parameter are local only
- The parameter is a new variable, initialised by the argument

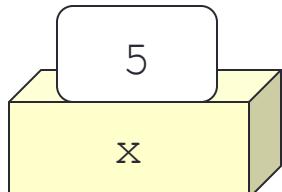
```
private void f(int a)
{a = 66;}
```

# Changes to parameters are usually lost

- When we call a method with a parameter of a primitive type, any updates to that parameter are local only
- The parameter is a new variable, initialised by the argument

```
private void f(int a)
{a = 66;}
```

```
int x = 5;
```



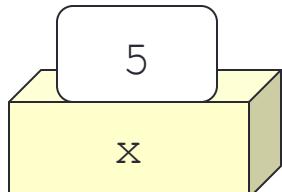
# Changes to parameters are usually lost

- When we call a method with a parameter of a primitive type, any updates to that parameter are local only
- The parameter is a new variable, initialised by the argument

```
private void f(int a)  
{a = 66;}
```

5

```
int x = 5;  
System.out.println(x);
```



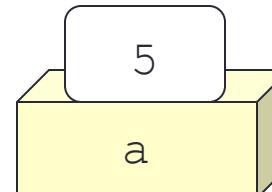
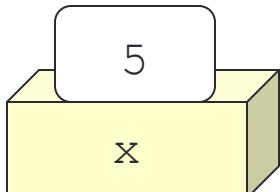
# Changes to parameters are usually lost

- When we call a method with a parameter of a primitive type, any updates to that parameter are local only
- The parameter is a new variable, initialised by the argument

```
private void f(int a)  
{a = 66;}
```

5

```
int x = 5;  
System.out.println(x);  
f(x);
```



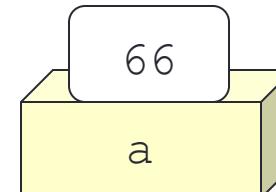
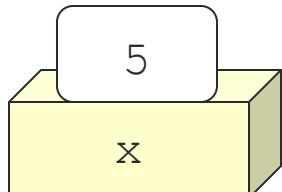
# Changes to parameters are usually lost

- When we call a method with a parameter of a primitive type, any updates to that parameter are local only
- The parameter is a new variable, initialised by the argument

```
private void f(int a)  
{a = 66;}
```

5

```
int x = 5;  
System.out.println(x);  
f(x);
```



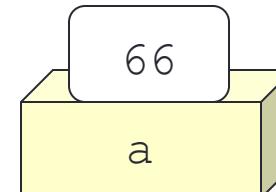
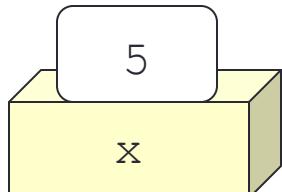
# Changes to parameters are usually lost

- When we call a method with a parameter of a primitive type, any updates to that parameter are local only
- The parameter is a new variable, initialised by the argument

```
private void f(int a)
{a = 66;}
```

5  
5

```
int x = 5;
System.out.println(x);
f(x);
System.out.println(x);
```



# Changes to parameters are usually lost

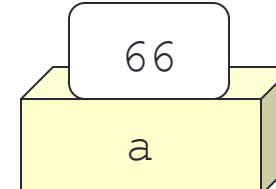
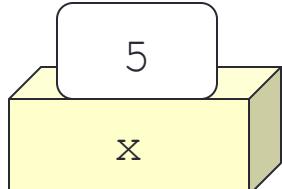
- When we call a method with a parameter of a primitive type, any updates to that parameter are local only
- The parameter is a new variable, initialised by the argument

```
private void f(int a)  
{a = 66;}
```

5  
5

```
int x = 5;  
System.out.println(x);  
f(x);  
System.out.println(x);
```

x is unchanged by f



# But arrays are persistent

- When we call a method with a parameter of an object type, the parameter is a new variable  
*but it refers to the same object*

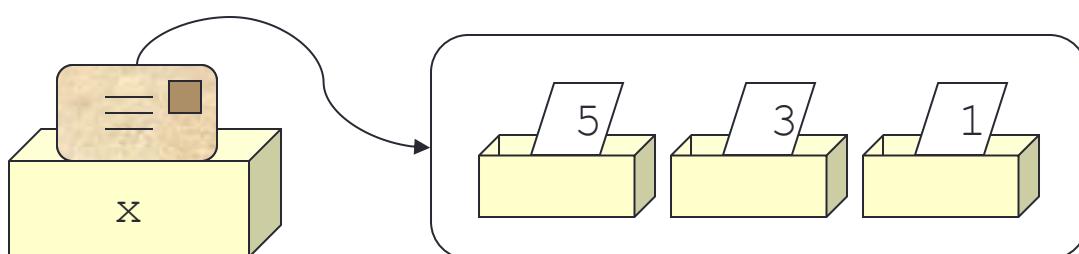
```
private void g(int[] a)  
{a[0] = 66; }
```

# But arrays are persistent

- When we call a method with a parameter of an object type, the parameter is a new variable  
*but it refers to the same object*

```
private void g(int[] a)  
{a[0] = 66;}
```

```
int[] x = {5, 3, 1};
```



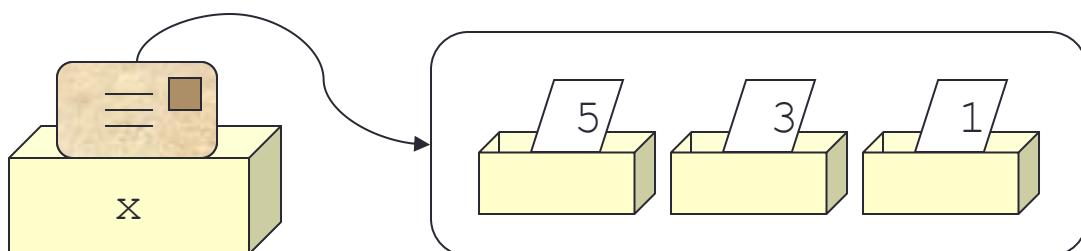
# But arrays are persistent

- When we call a method with a parameter of an object type, the parameter is a new variable  
*but it refers to the same object*

```
private void g(int[] a)  
{a[0] = 66;}
```

5

```
int[] x = {5,3,1};  
System.out.println(x[0]);
```



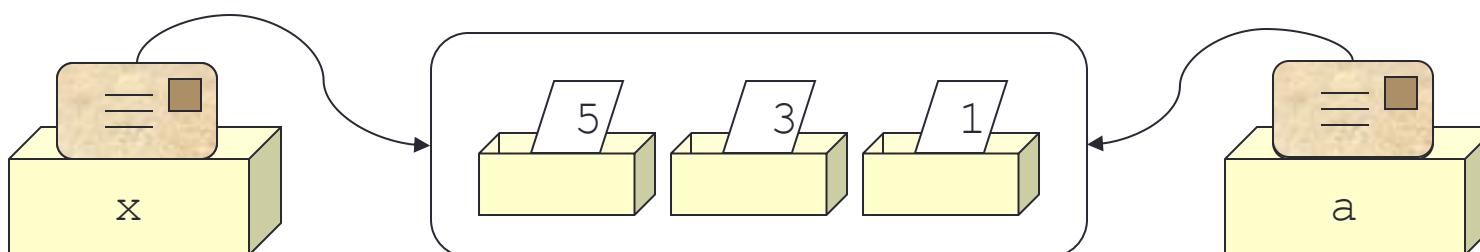
# But arrays are persistent

- When we call a method with a parameter of an object type, the parameter is a new variable  
*but it refers to the same object*

```
private void g(int[] a)  
{a[0] = 66;}
```

5

```
int[] x = {5,3,1};  
System.out.println(x[0]);  
g(x);
```



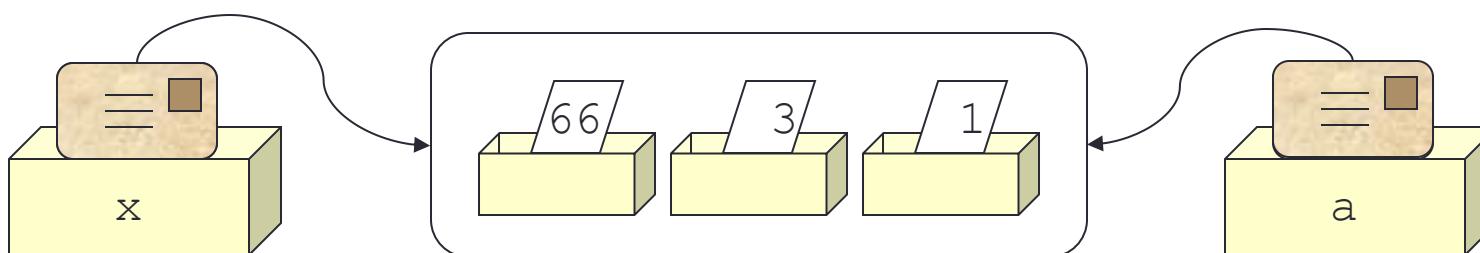
# But arrays are persistent

- When we call a method with a parameter of an object type, the parameter is a new variable  
*but it refers to the same object*

```
private void g(int[] a)  
{a[0] = 66;}
```

5

```
int[] x = {5,3,1};  
System.out.println(x[0]);  
g(x);
```



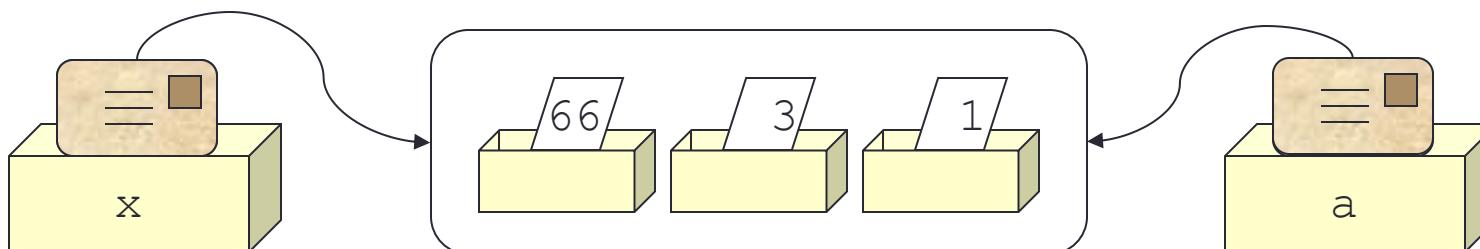
# But arrays are persistent

- When we call a method with a parameter of an object type, the parameter is a new variable  
*but it refers to the same object*

```
private void g(int[] a)  
{a[0] = 66;}
```



```
int[] x = {5,3,1};  
System.out.println(x[0]);  
g(x);  
System.out.println(x[0]);
```



# But arrays are persistent

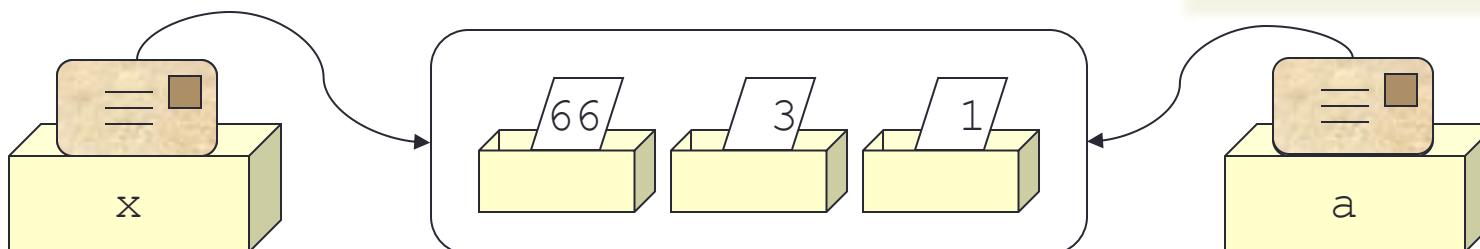
- When we call a method with a parameter of an object type, the parameter is a new variable  
*but it refers to the same object*

```
private void g(int[] a)
{a[0] = 66;}
```

5
66

```
int[] x = {5,3,1};
System.out.println(x[0]);
g(x);
System.out.println(x[0]);
```

x is unchanged by g  
but x [0] is changed



# Back to the earlier example

- Sometimes instead of creating a new array, we want to update the old one

```
private void sums(int[] a)
{
    for (int i = 1; i < a.length; i++)
    {
        a[i] = a[i-1] + a[i];
    }
}
```

- So the method doesn't return anything, but its changes persist through aliasing

English Teachers: You will almost never use the semicolon.

Java programmers:



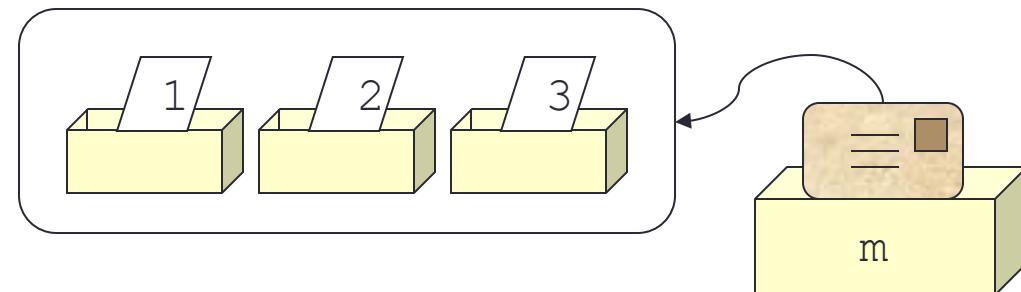
# Equality over objects

- Be careful with equality over arrays and objects

# Equality over objects

- Be careful with equality over arrays and objects

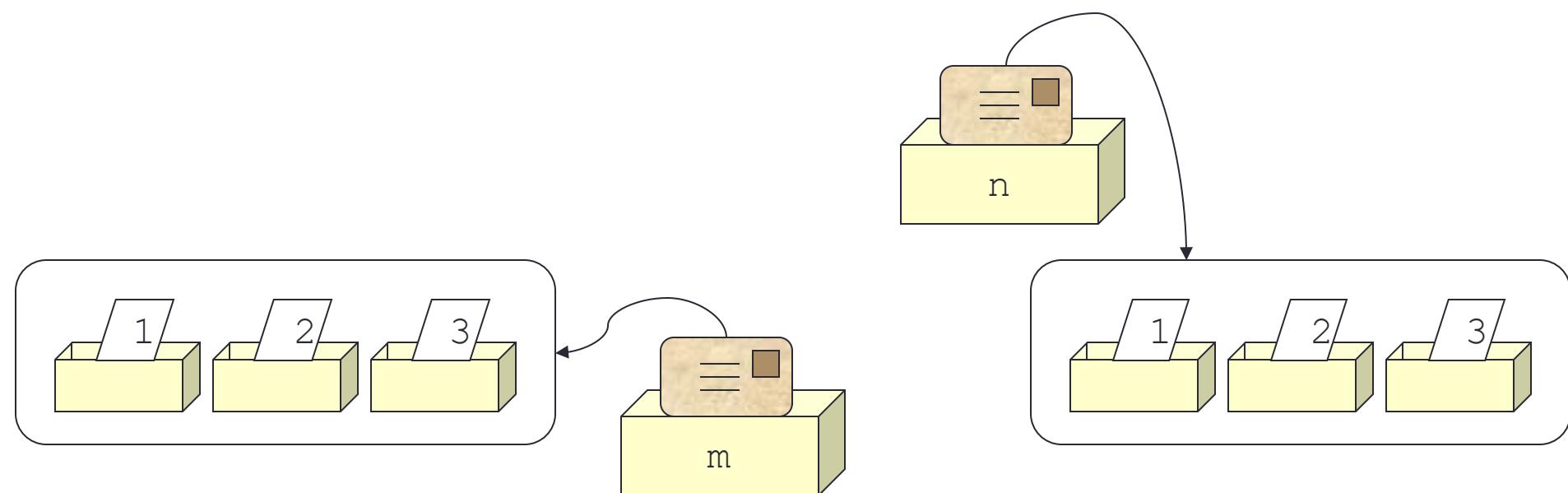
```
int[] m = {1, 2, 3};
```



# Equality over objects

- Be careful with equality over arrays and objects

```
int[] m = {1,2,3};  
int[] n = {1,2,3};
```

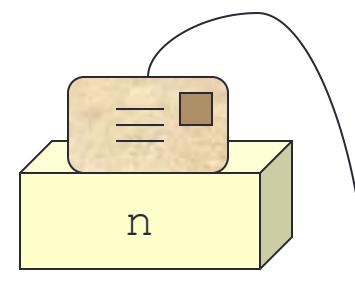
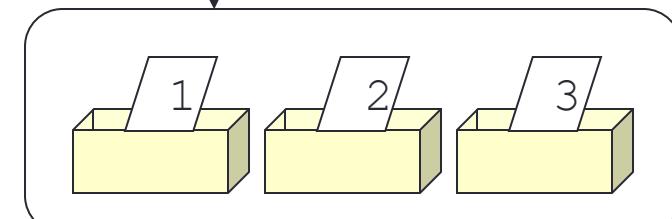
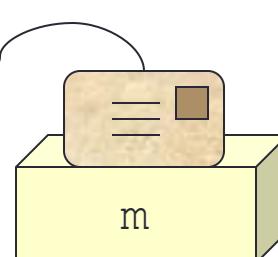
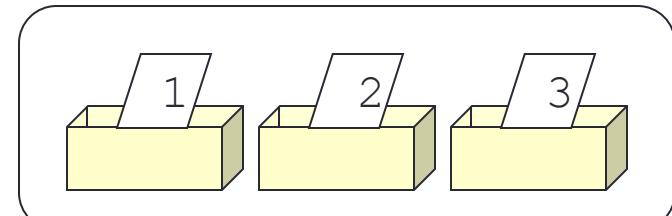
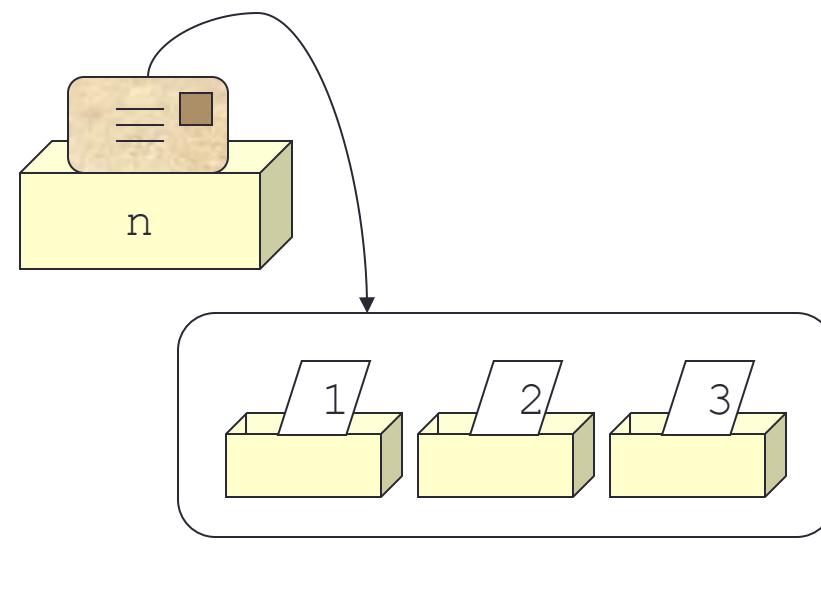


# Equality over objects

- Be careful with equality over arrays and objects

```
int[] m = {1,2,3};  
int[] n = {1,2,3};  
System.out.println(m == n);
```

false

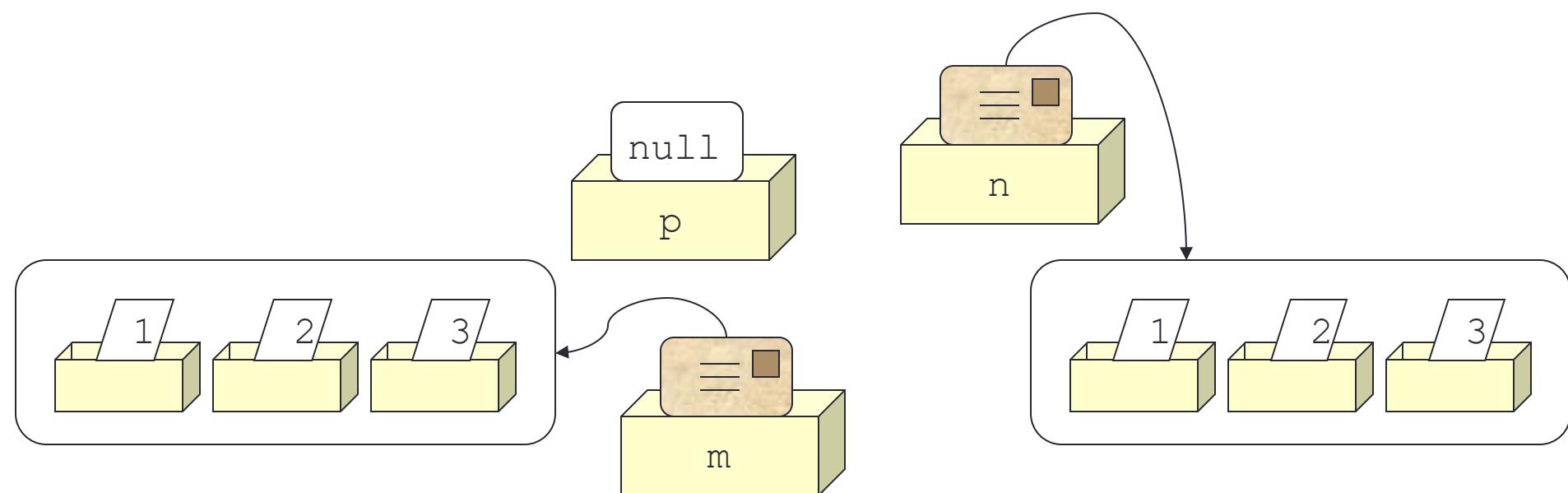


# Equality over objects

- Be careful with equality over arrays and objects

```
int[] m = {1,2,3};  
int[] n = {1,2,3};  
System.out.println(m == n);  
int[] p;
```

false

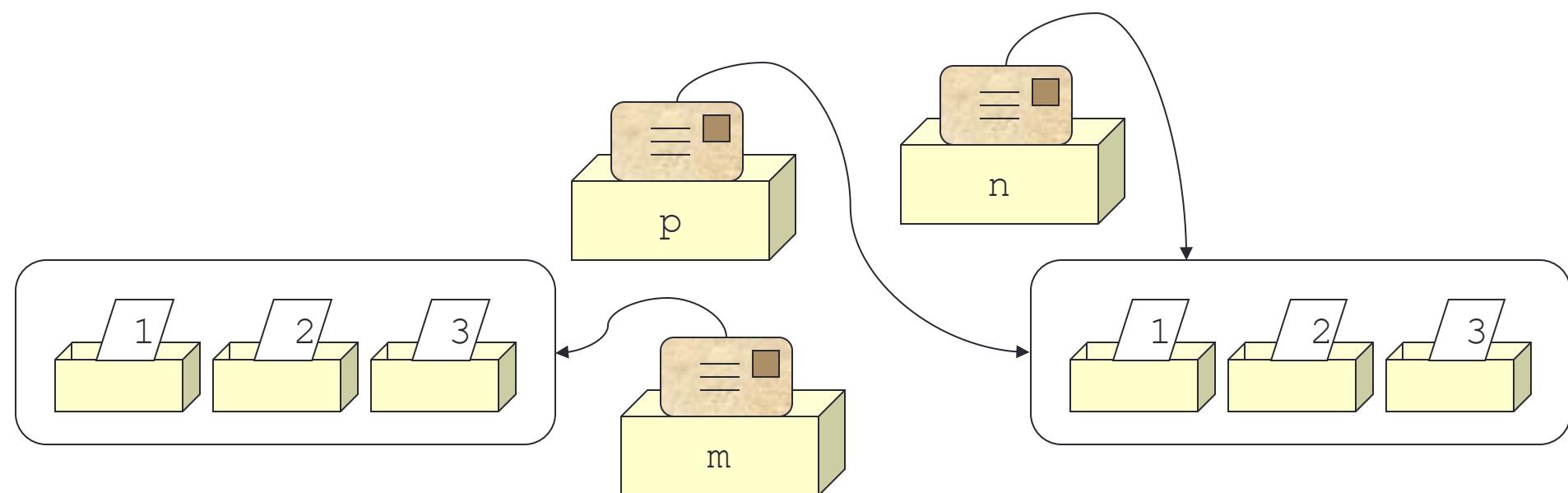


# Equality over objects

- Be careful with equality over arrays and objects

```
int[] m = {1,2,3};  
int[] n = {1,2,3};  
System.out.println(m == n);  
int[] p;  
p = n;
```

false

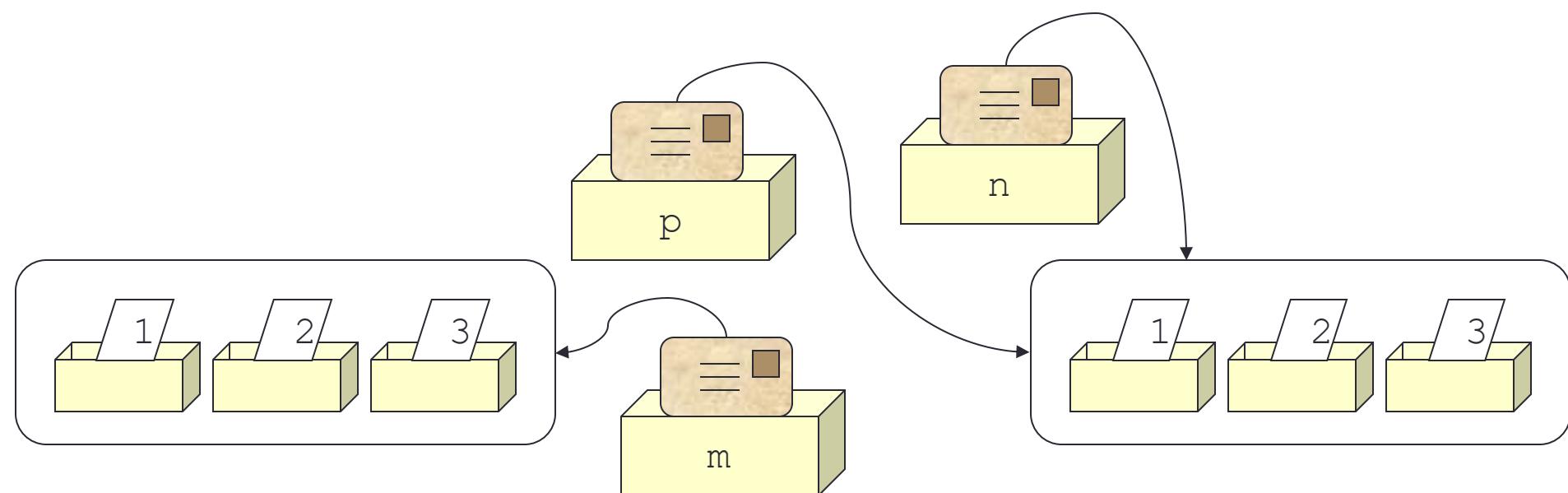


# Equality over objects

- Be careful with equality over arrays and objects

```
int[] m = {1,2,3};  
int[] n = {1,2,3};  
System.out.println(m == n);  
int[] p;  
p = n;  
System.out.println(p == n);
```

false  
true

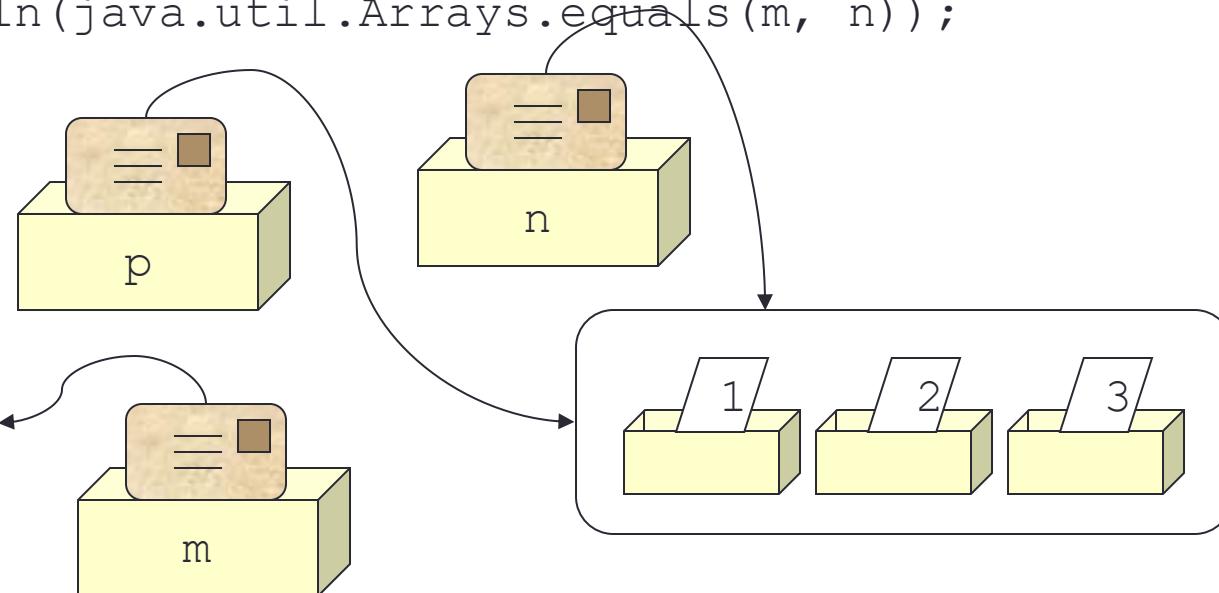


# Equality over objects

- Be careful with equality over arrays and objects

```
int[] m = {1,2,3};  
int[] n = {1,2,3};  
System.out.println(m == n);  
int[] p;  
p = n;  
System.out.println(p == n);  
System.out.println(java.util.Arrays.equals(m, n));
```

false  
true  
true



# Arrays of objects

- The arrays we have seen so far had elements of a primitive type
  - `int[]`, `double[]`, `boolean[]`, etc.
- But arrays can also hold objects
  - e.g. `Student[]`
- Arrays can also be two-dimensional
  - e.g. `int[][]`
- Arrays can also be three-dimensional, or more
  - But we won't get to that in CITS1001

# Arrays of objects

- When using an array with elements of primitive type, there are two steps involved
  - *Declare* the variable to refer to the array
  - *Create* the space for the array elements using `new`
- When using an array with elements of object type, there are *three* steps involved
  - *Declare* the variable to refer to the array
  - *Create* the space for the array elements using `new`
  - *Populate* the array with objects by ***repeatedly using*** `new`, often in a loop

# A Student class

```
public class Student {  
    private String studentNumber;  
    private int mark;  
  
    public Student(String studentNumber, int mark)  
    {this.studentNumber = studentNumber;  
     this.mark = mark; }  
  
    public String getStudentNumber()  
    {return studentNumber;}  
  
    public int getMark()  
    {return mark; }  
}
```

A skeleton version of  
a possible Student  
class in a student  
records system

# Creating a unit list

```
Student[] unit;
```

*Declare*

```
unit = new Student[numStudents];
```

*Create*

```
unit[0] = new Student("042371X", 64);
```

```
unit[1] = new Student("0499731", 72);
```

```
unit[2] = new Student("0400127", 55);
```

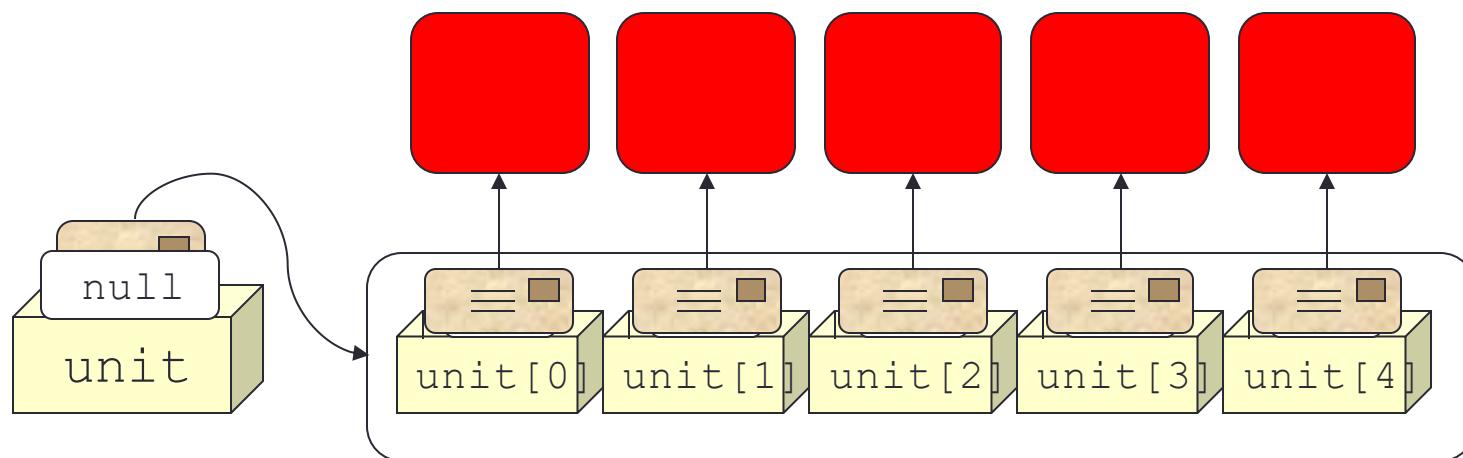
*Populate*

...

```
unit[numStudents-1] = new Student("0401332", 85);
```

# The three steps

- Declare
- Create
  - Using `new`
- Populate (repeatedly)
  - *Each object needs `new` to be executed*



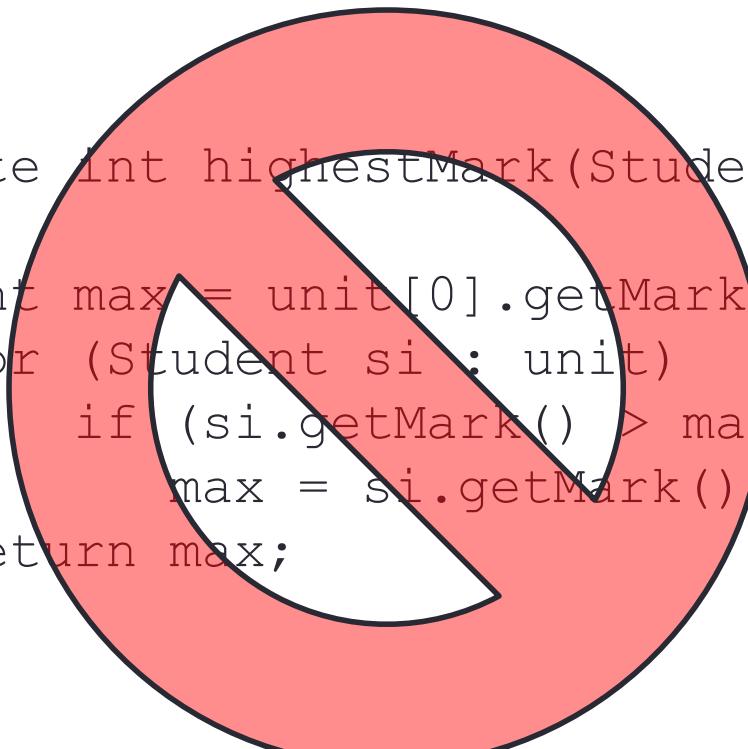
# Using arrays of objects

- Using an array of objects is the same as using any array
  - The syntax is the same, and elements can be used in the same ways
  - You just need to remember that the elements are objects!
- This method returns the best-performing Student in unit

```
private Student top(Student[] unit)
{
    Student top = unit[0];
    for (Student si : unit)
        if (si.getMark() > top.getMark())
            top = si;
    return top;
}
```

# What if we want to find the highest mark?

- We could copy-and-paste top, and edit the details to create a new method



```
private int highestMark(Student[] unit)
{
    int max = unit[0].getMark();
    for (Student si : unit)
        if (si.getMark() > max)
            max = si.getMark();
    return max;
}
```

**No!!**

# Much better to use the existing method!

- ***Do not*** write another looping method!
  - Use the already-written functionality

```
private int highestMark(Student[] unit)
{
    return top(unit).getMark();
}
```

**find the best Student**

**and get their mark**

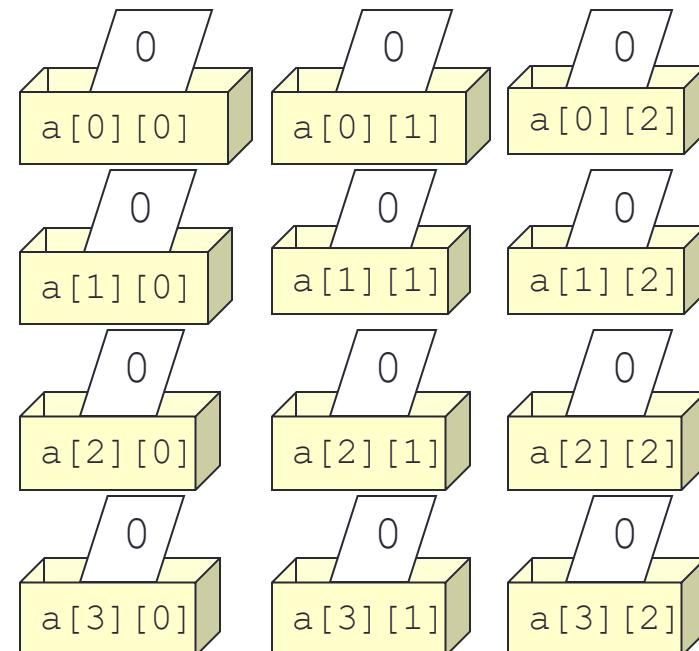
# 2D arrays

- We sometimes need an array with more than one dimension

```
int[][] a = new int[4][3];
```

This creates an array with four “rows” and three “columns”

The “row” index ranges from 0–3 and the “column” index from 0–2





**MATRIX**

---

**2D-ARRAY**

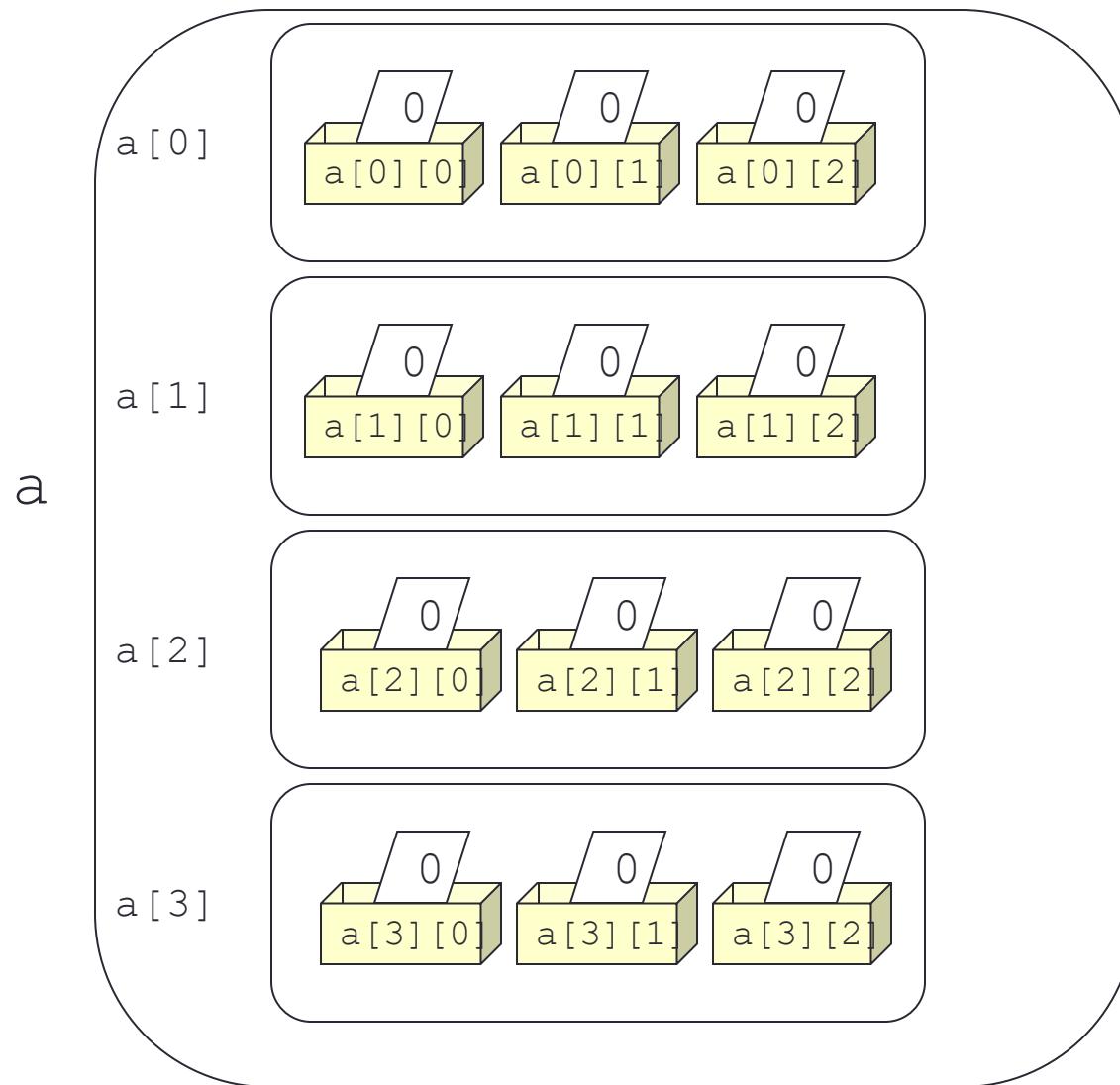
---

**LIST OF LISTS**

# 2D arrays contd.

- This is technically an array where each element is itself an array
  - `a` is a valid variable, representing the entire thing
  - `a[0]` is a valid variable, representing the first “row”
  - `a[0][2]` is a valid variable, representing the last element in the first “row”
- But notice there is no single variable representing a “column” of the array
  - A column is not a single object

# 2D arrays contd.



## 2D arrays contd.

- Given that an array is an object, this is really just a special case of an array of objects
- The single statement

```
int[][] a = new int[4][3];
```

declares the array variable `a`, creates the array,  
*and populates it!*

# 2D arrays examples

- A 2D array is normally processed with two nested for loops

```
private int sum(int[][] a) {  
    int sum = 0;  
    for (int i = 0; i < a.length; i++)  
        for (int j = 0; j < a[i].length; j++)  
            sum += a[i][j];  
    return sum;  
}
```

**number of rows**

**number of columns in Row i**

# 2D arrays examples

- Suppose we want to return the index of the first row here that contains any element that is true

```
private int firsttrue(boolean[][] a)
{
    for (int i = 0; i < a.length; i++)
        for (int j = 0; j < a[i].length; j++)
            if (a[i][j])
                return i;
    return -1;
}
```

# 2D arrays example

- We will illustrate the use of 2D arrays further with a case study of The Game of Life
- [http://en.wikipedia.org/wiki/Conway's\\_Game\\_of\\_Life](http://en.wikipedia.org/wiki/Conway's_Game_of_Life)
  - In the next lecture...
  - And there's plenty of practice in the lab too

# CITS1001

## 12. ENUMERATED TYPES

---

*Objects First With Java, Chapter 8*

# Lecture essentials

- Some data has values in a restricted discrete range
  - e.g. days of the week, months of the year
- Representing this kind of data in built-in types is dangerous
  - Inadvertent or deliberate misuse is easy
- We can represent this data best with enumerated types
  - Defined in Java through a special type of class enum
- enum comes with several built-in facilities

# Declaring a variable

- Whenever you declare a variable in a program, you should always have a sense of what information that variable will be used to store
  - In Class Structure we called this the *meaning* of the variable
- For example

```
int profit; // holds the annual profit of a company  
int price; // holds the price of a car  
int mark; // holds someone's mark in a unit  
int date; // holds the day of the month
```

- These are all int variables, but the ways they are used in a program will be very different

# The range of a variable

- Also the likely *ranges* of these variables are very different
  - profit can be any value, positive or negative
  - price can only be positive
  - mark can only be in the range 0..100
  - date can only be in the range 1..31
- The ranges of some data are even shorter
  - Months of the year?
  - Days of the week?
  - Suits in a deck of cards?
- How should we represent these restrictions in a program?
  - In general this is a tough question, but for some kinds of data there is a good solution

# Discrete data types

- We say that a data type is *discrete* when it has a list of possible values that can't be broken down any further
  - e.g. int 1, 2, 3, ...
  - e.g. char 'a', 'b', 'c', ...
  - e.g. boolean true, false
- Contrast this for example with types whose values have sub-structure in some sense
  - e.g. String, ArrayList, double

# Discrete data with a restricted range

- Some data that we want to represent is discrete with only a *small number of possible values*
- e.g. days of the week
  - Monday, Tuesday, Wednesday, etc.
- e.g. months of the year
  - January, February, March, etc.
- e.g. suits in a deck of cards
  - Spades, Hearts, Diamonds, Clubs
- e.g. pieces on a chess board
  - Pawn, rook, knight, bishop, queen, king
- Many, many others

# Discrete data with a restricted range

- How can we represent this kind of data in a Java program?
  - There are two common ways
- Using int
  - Choose a number to represent each value,  
e.g. Monday = 1, Tuesday = 2, Wednesday = 3,  
Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7
- Using String
  - Use the names (or parts of them),  
e.g. "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"
- Such mappings are often called *encodings*

# Discrete data using int

- This method returns the number of hours worked by a casual tutor on a given day of the week

```
public int noOfHours(int day)
{
    if (day == 1) return 2;
    if (day == 2) return 5;
    if (day == 3) return 3;
    if (day == 4) return 6;
    if (day == 5) return 6;
    return 0;
}
```

# Discrete data using String

- This method returns the number of hours worked by a casual tutor on a given day of the week

```
public int noOfHours(String day)
{
    if (day == "Mon") return 2;
    if (day == "Tue") return 5;
    if (day == "Wed") return 3;
    if (day == "Thu") return 6;
    if (day == "Fri") return 6;
    return 0;
}
```

# Problems

- This approach has at least four serious problems
- What if a programmer or user enters an ‘illegal’ int/String?
  - e.g. 8, or “Thi”?
  - All methods must be written to allow for this possibility
- Will programmers and users remember the encoding?
  - e.g. was Monday 0, or 1?
  - e.g. was Thursday “Thursday”, or “Thu”, or “THU”?
- It is easy for programmers to inadvertently swap variables or values that are being used for different encodings
  - e.g. if you’re using `int d` for days and `int m` for months
- There’s nothing to stop programmers (accidentally or deliberately) using type-correct operations on days
  - e.g. adding `int` days together
  - e.g. taking a substring of a `String` day

# Problems

- One major issue with these problems is that they all become apparent only during execution
  - So code goes wrong at the worst possible time
  - Testing can achieve only so much
- Much better would be an approach which restricts programmers at compile time
  - We want these sorts of errors to be caught by syntax-checking or type-checking
  - Then they can be fixed before execution

# A better approach

- Better is to define a brand new type to represent days of the week, with **new values unique to that type**
- Java provides a special (and very simple!) type of class called the `enum`

```
public enum Day  
{MON, TUE, WED, THU, FRI, SAT, SUN}
```

- Day is now a new type in the program
- The values are accessed as e.g. `Day.MON`, `Day.TUE`
  - They are written using capitals because they are constants



Jay Phelps  
@\_jayphelps

```
enum Boolean {  
    true,  
    false,  
    maybe  
}
```

[Traduzir Tweet](#)

13:03 · 22 nov 19 · [Twitter Web App](#)

---

45 Retweets 391 Curtidas

---



# Discrete data using Day

- This method returns the number of hours worked by a casual tutor on a given day of the week

```
public int noOfHours(Day day)
{
    if (day == Day.MON) return 2;
    if (day == Day.TUE) return 5;
    if (day == Day.WED) return 3;
    if (day == Day.THU) return 6;
    if (day == Day.FRI) return 6;
    return 0;
}
```

# Problems solved!

- Illegal values (e.g. 8, “Thi”) are now impossible
  - They would cause a type error
- There is no encoding
  - The data is implicit in the names of the values
- Inadvertent swaps are now possible only with other uses of Day, not with a separate enum like Month
  - Many fewer opportunities for mistakes
- The only operations that can be performed on Days are those defined in the program
  - The programmers who manipulate the values retain control over the new type

# enum comes with built-in facilities

- Days can be compared for equality and inequality

```
Day d1, d2;  
if (d1 == d2) {} {}
```

- They can be added to Strings

```
Day d = Day.TUE;  
String s = "I love " + d; // sets s to "I love TUE"
```

- The range can be accessed easily...

- Day.values() denotes the Day[] array  
{Day.MON, Day.TUE, ..., Day.SUN}

- ...so they can be processed using a for-each loop

```
for (Day d : Day.values()) {}
```

- They can be ordered using the method ordinal()

```
int x = d.ordinal(); // sets x to 1
```

# The End

# CITS1001

## 13. THE GAME OF LIFE – A LARGER CASE STUDY

---

### PART 1

# Lecture essentials

- Software design
  - Maintenance, coupling, cohesion
- Case study: the Game of Life
  - Definition
  - Design
  - Implementation
  - Performance issues
- References
  - [https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)
  - <https://bitstorm.org/gameoflife/>
  - (and hundreds of other web-sites)

# Software maintenance

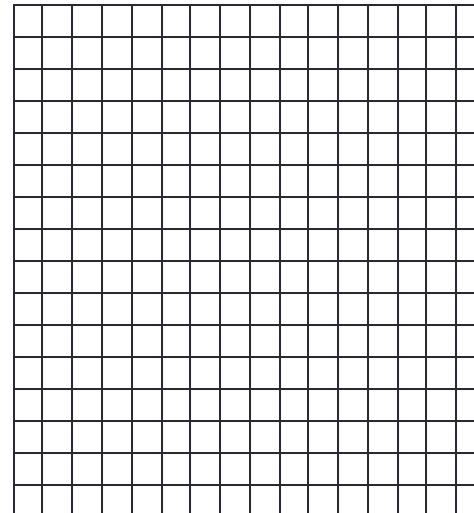
- Software is not like a novel that is written once and then is left unchanged for the ages
- After it is “finished”, software is
  - Corrected – any bugs might be fixed
  - Extended – functionality might be added
  - Ported – it might be moved to different systems
  - Adapted – it might be changed to use new hardware or software
  - Translated – it might be re-written in other languages
- This activity is collectively known as *maintenance*
  - This work is done by different people at different times in varying roles, sometimes over years or even decades
- Software either changes and adapts, or it dies
  - Making software easy to understand makes it easier to modify reliably, which facilitates this change

# Software design principles

- We have already come across abstraction and modularization
- Two other important concepts in software design are *coupling* and *cohesion*
- Coupling refers to the links between different parts of a program
  - If two classes depend closely on many details of each other's workings, we say that they are *tightly-coupled*
  - We aim for loose coupling; this allows different parts of the program to be worked-on independently
- Cohesion refers to the number and diversity of tasks that a single class is responsible for
  - If a class is responsible for a single logically-contained task, we say that it has *high cohesion*
  - We aim for high cohesion; this promotes reliability and understandability

# The Game of Life

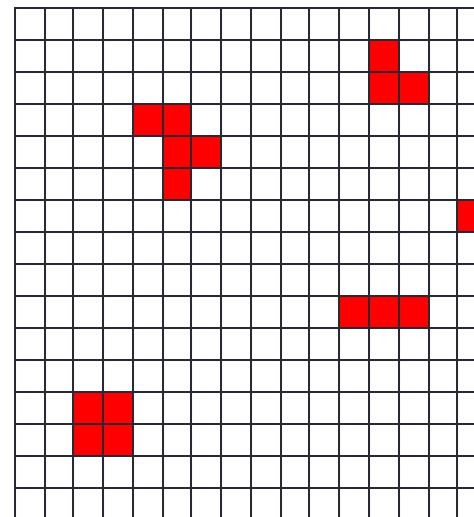
- Invented by John Conway\* in 1970
- An example of a *cellular automaton*
- The game is played on a rectangular grid of *cells*



\*John Conway died from COVID-19 on 11 April 2020

# A model world

- The grid is a simple *model world* where at any given moment, each cell is either occupied or vacant
- The initial configuration might be chosen by the user, or it might be set up randomly

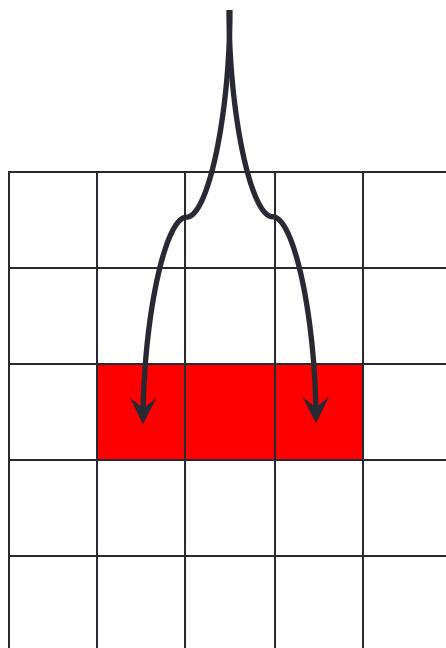


# Births and deaths

- In this model world, time passes in discrete steps known as *generations*
- The beginning of time is *Generation 0*
- At each time step, the occupants of the model world live or die according to three rules:
  - An individual with zero or one neighbours dies of *loneliness*
  - An individual with four or more neighbours dies of *overcrowding*
  - A new individual is *born* in any vacant cell with exactly three neighbours
- The neighbours of a cell  $C$  are the occupants of the eight cells surrounding  $C$

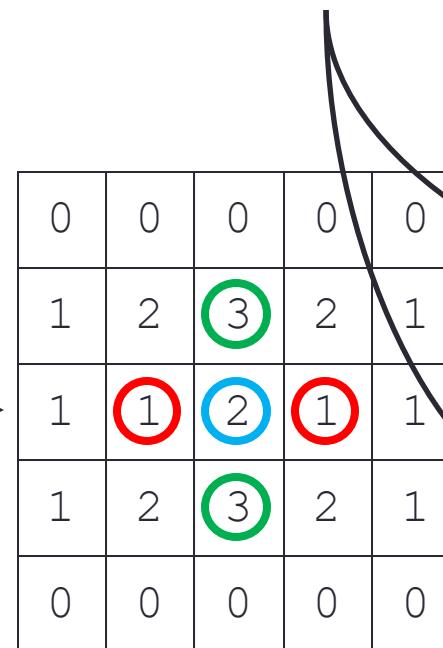
# An example

The occupants of these cells die of loneliness



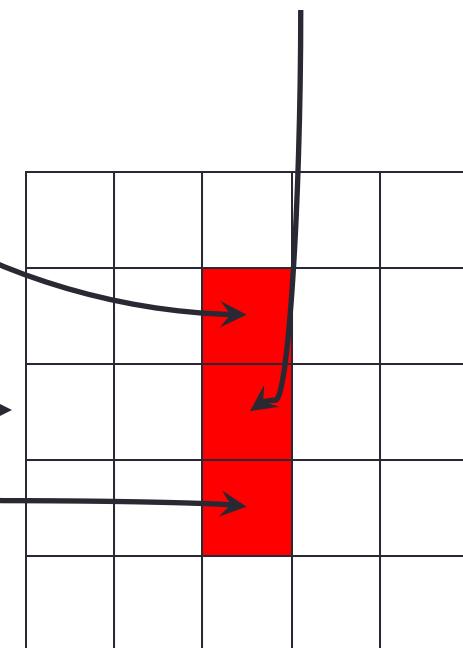
Generation  $k$

These two are new-born “babies”



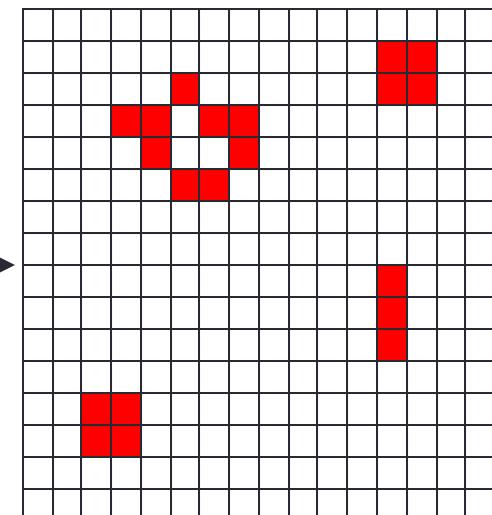
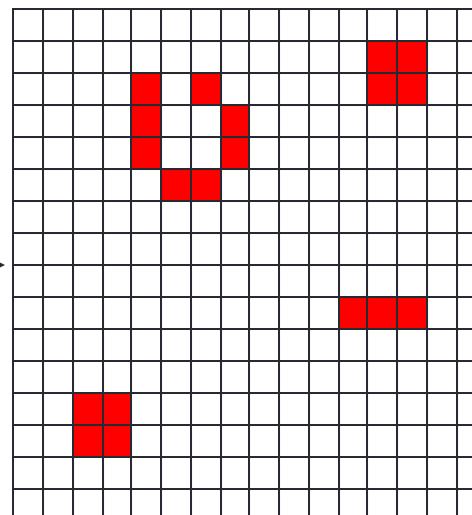
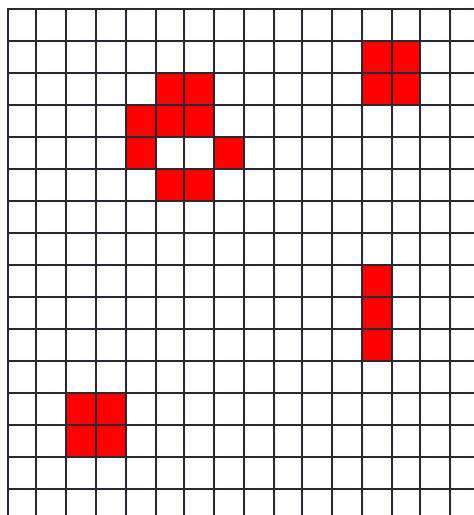
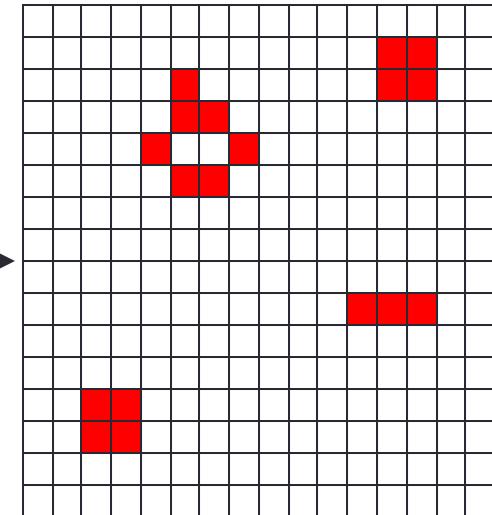
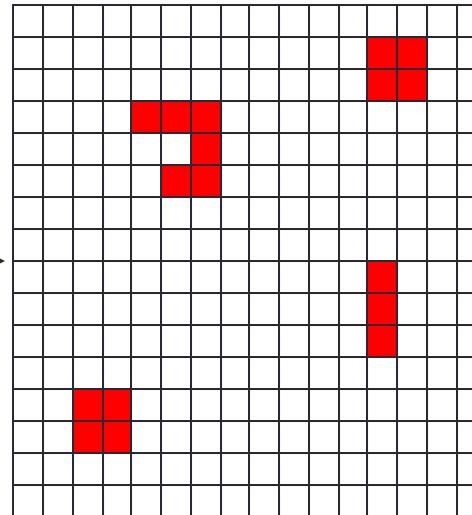
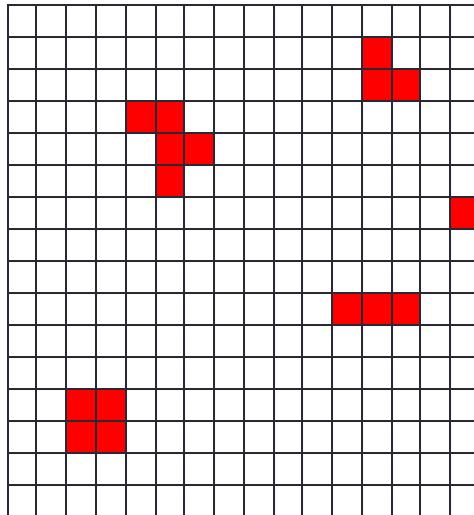
No. of neighbours  
of each cell

This one is a survivor



Generation  $k+1$

# Generational change

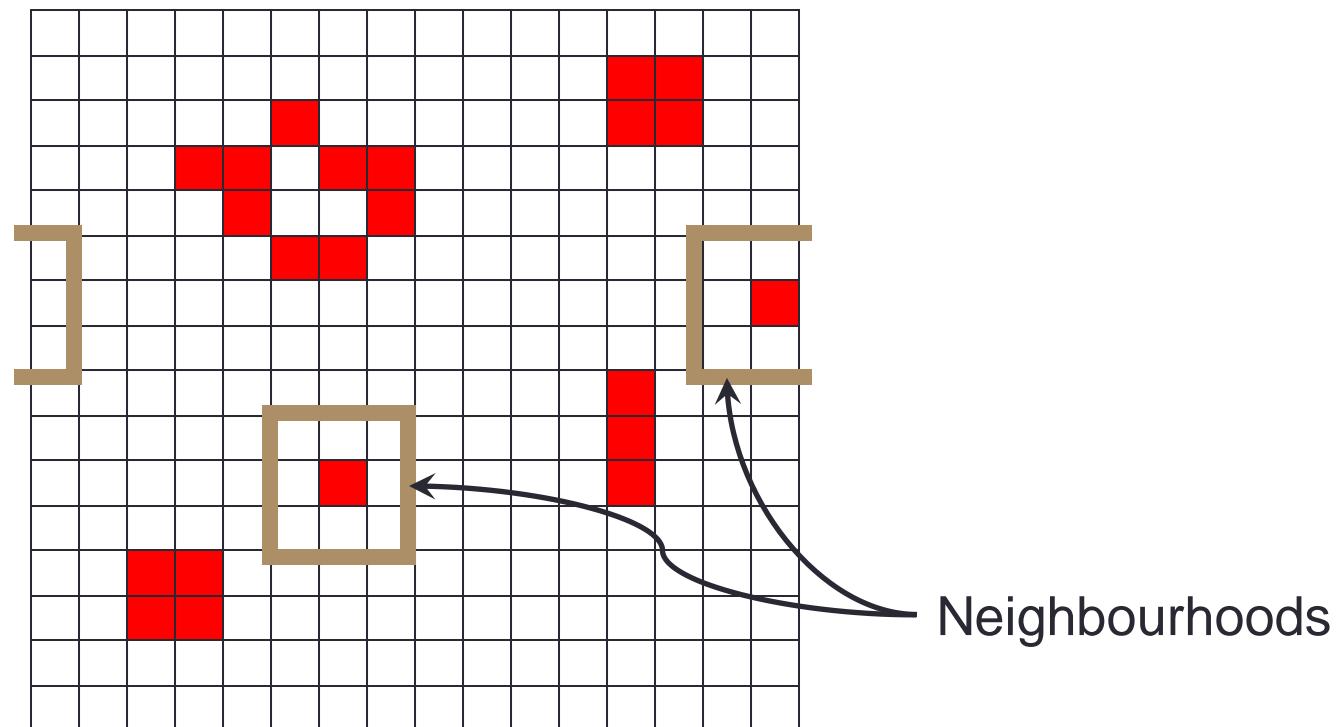


# Check out these two links

- Wikipedia
  - [https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)
- Bitstorm
  - <https://bitstorm.org/gameoflife/>

# At the edges

- The edges of the world are assumed to “wrap around”
    - So e.g. cells on the right edge are neighbours of cells on the left edge
  - The map is actually a flat depiction of a torus-shaped world



# Program specification

- We want to write a program that allows a user to experiment with The Game of Life
  - To set up specific configurations
  - Or to have a configuration provided for them
  - To run some (controllable) number of generations
  - Or simply to animate a configuration
- Other possibilities might occur to you
  - e.g. to animate a configuration and to test it for stability or for other properties
  - e.g. to compare two configurations in some way
- The program may need to be changed as new possibilities come up...
  - Hence the need for good design

# Program design

- The program must do three jobs
  - Maintain an *internal* representation of the map of the world
  - Turn this representation into a display that users can interact with
  - Render this display on the screen
- These jobs will correspond to our three classes
  - Life
  - LifeViewer
  - SimpleCanvas
- One job is allocated to one class
  - High cohesion and loose coupling

# Separate the responsibilities

- The `Life` class should
  - Maintain the internal map of the world
  - Update to the next generation when requested
  - Provide access to the map for client use
- The `LifeViewer` class should
  - Create a `SimpleCanvas` on which to visualise the map
  - Turn each generation into an appropriate image
  - Animate a client-specified number of generations
- The `SimpleCanvas` class should
  - Talk to the operating system to display the visualisation
  - Update the screen when required
- We will look at these three classes in turn

# Life instance variables

- We use a 2D array of booleans for the map of the world
  - The value of the `[i] [j]` entry in the array `map` indicates whether the  $(i,j)$  cell is occupied or vacant
  - We also store the dimensions of the map, for convenience

```
public class Life {  
    private boolean[][] map;  
    private int width, height;  
  
    // constructors & methods  
}
```

# Why no enum?

- In the last lecture we learned how to use enum classes to avoid these kinds of encodings
  - So why not use something like this?

```
public enum Cell  
{OCCUPIED, VACANT}
```

- An enum **is not so beneficial here**
  - Illegal values are impossible when boolean **is** used
  - The encoding is trivial
  - Transposition of variables is impossible
  - Inadvertent operators are unlikely

# Break

- <https://youtu.be/C2vgICfQawE>

# The first constructor

- This constructor allows a user to create an initial world with any pattern of occupied and vacant cells that they want

```
public Life(boolean[][][] initial) {  
    map = initial;  
    width = map.length;  
    height = map[0].length; // initial is assumed to be  
} // non-null and rectangular
```

# The second constructor

- This constructor allows a user to specify the dimensions of the world and to have a random configuration created
  - Note the use of `this`

```
public Life(int width, int height, double probability) {  
    map = new boolean [width] [height];  
    this.width = width;  
    this.height = height;  
    initializeMap (probability);  
}
```

# Private helper method for initialisation

- If the user specifies 0.2 as the probability,  
approx. 20% of the cells should be occupied initially
  - Note the use of `Math.random`

```
private void initializeMap(double probability) {  
    for (int i = 0; i < width; i++)  
        for (int j = 0; j < height; j++)  
            map[i][j] = Math.random() < probability;  
}
```

# The “default” constructor

- It is polite to give a constructor that is easy to use for testing purposes
  - Basically the second constructor with programmer-chosen values

```
public Life() {  
    this(128, 128, 0.1);  
}
```

# Making the next generation

```
public void nextGeneration() {  
    boolean[][] nextMap = new boolean[width][height];  
    for (int i = 0; i < width; i++)  
        for (int j = 0; j < height; j++) {  
            int z = numNeighbours(i, j);  
            if (z == 2) nextMap[i][j] = map[i][j];  
            else         nextMap[i][j] = z == 3;  
        }  
    map = nextMap;  
}
```

# Code dissection

```
boolean [ ] [ ] nextMap = new boolean [width] [height];
```

- This creates the space to hold the “next generation”
  - Same dimensions as the current map
- We cannot do the updates “in place”, because notionally all cells update simultaneously
  - If we started updating Row  $k$  in place, the updating for Row  $k+1$  would be wrong
  - It needs to see the “old” values for Row  $k$

# Code dissection

```
int z = numNeighbours(i, j)  
if (z == 2) nextMap[i][j] = map[i][j];  
else          nextMap[i][j] = z == 3;
```

We haven't defined  
this method yet

- For the  $(i,j)$  cell (inside the loop)
  - Calculate its number of neighbours
  - Implement the rules of the game
- Check that you understand how this code implements the rules from earlier

# Code dissection

```
map = nextMap;
```

- Finally we make `map` point to the newly-created object pointed to by `nextMap`
- Everything is now correctly updated for the next generation

# Counting the neighbours

```
private int numNeighbours(int i, int j) {  
    int n = 0;  
    int ip = (i + 1) % width;  
    int im = (width + i - 1) % width;  
    int jp = (j + 1) % height;  
    int jm = (height + j - 1) % height;  
    if (map[im][jm]) n++; if (map[im][j]) n++;  
    if (map[im][jp]) n++; if (map[i][jm]) n++;  
    if (map[i][jp]) n++; if (map[ip][jm]) n++;  
    if (map[ip][j]) n++; if (map[ip][jp]) n++;  
    return n;  
}
```

# Code dissection

```
int n = 0;  
<some updates in here>  
return n;
```

- n is used as an accumulating variable

# Code dissection

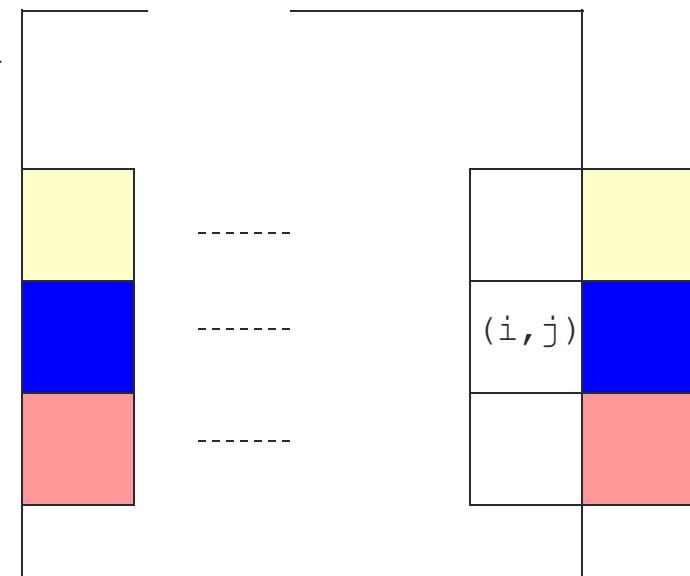
```
int ip = (i + 1)           % width;  
int im = (width + i - 1)   % width;  
int jp = (j + 1)           % height;  
int jm = (height + j - 1)  % height;
```

- **ip, im, jp, jm are the coordinates of the cells next to  $i, j$**

[i-1] [j-1]	[i] [j-1]	[i+1] [j-1]
[i-1] [j]	[i] [j]	[i+1] [j]
[i-1] [j+1]	[i] [j+1]	[i+1] [j+1]

# Code dissection

- The complication comes from dealing with wrap-around
- If  $i = \text{width}-1$ , the cell on its “right” has coordinate 0
  - But of course  $i+1$  gives  $\text{width}$
  - We can correct this with  $\% \text{ width}$
- When  $i = 0$ , the cell on its “left” is  $\text{width}-1$ 
  - But  $i-1$  gives  $-1$
  - We can’t correct this simply with  $\% \text{ width}$
  - So add  $\text{width}$  before doing  $\% \text{ width}$ !



# Code dissection

```
if (map[im][jm]) n++; if (map[im][j]) n++;  
if (map[im][jp]) n++; if (map[i][jm]) n++;  
if (map[i][jp]) n++; if (map[ip][jm]) n++;  
if (map[ip][j]) n++; if (map[ip][jp]) n++;
```

- These eight conditionals check the neighbours and update n

To be continued in Part 2

# CITS1001

## 13. THE GAME OF LIFE – A LARGER CASE STUDY

---

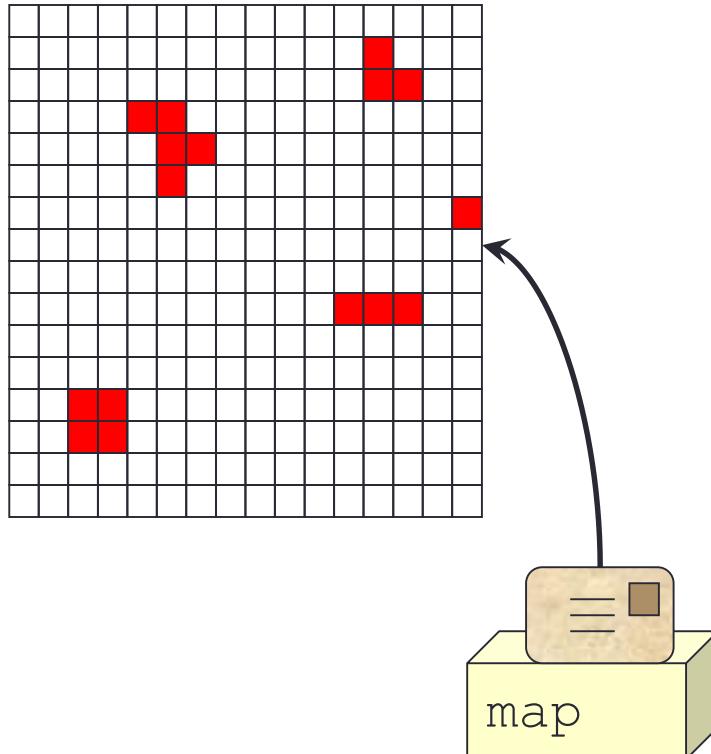
### PART 2

# Continue to

- Improve the Life class
- Implement the LifeViewer class

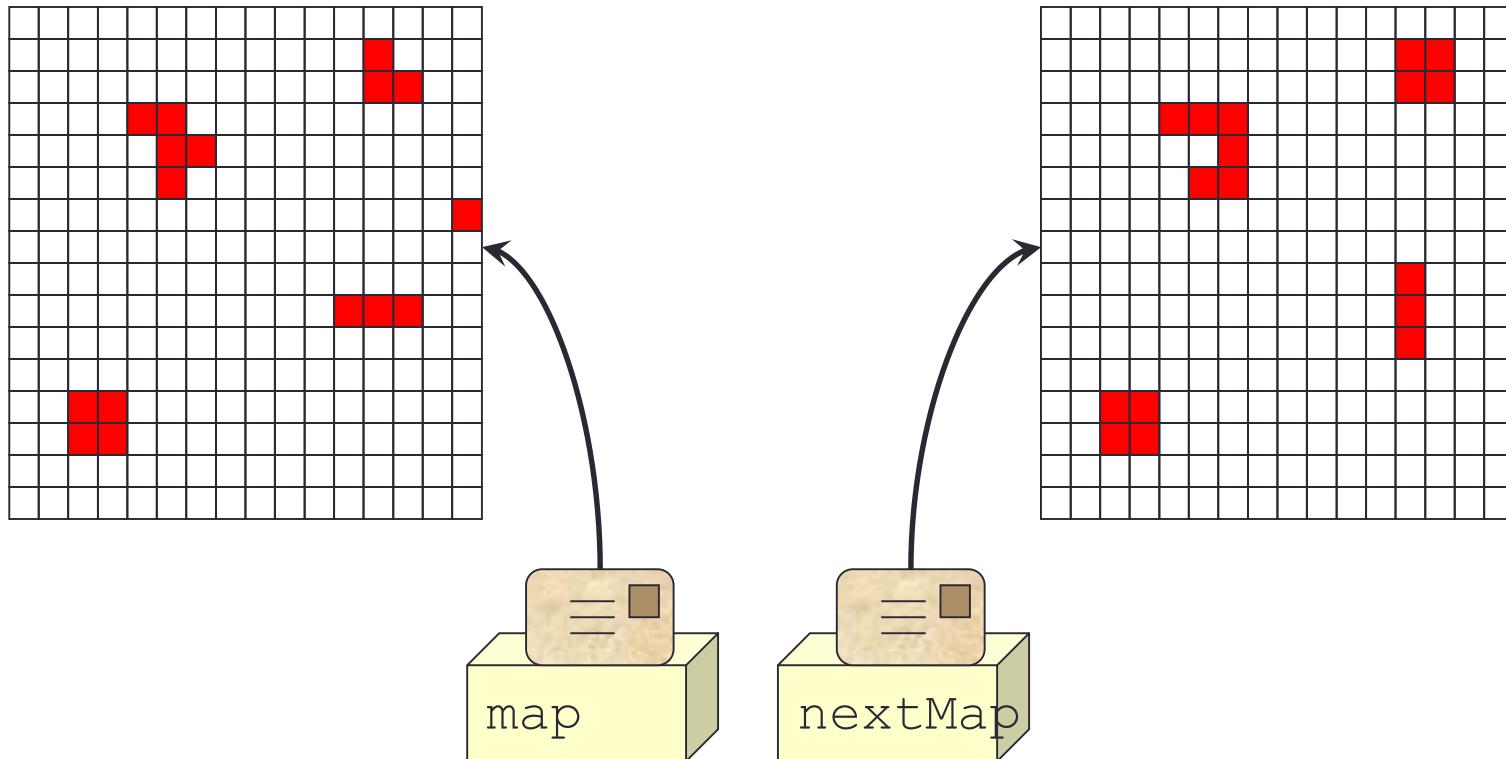
# A performance issue

- As written, this code consumes a lot of *memory*



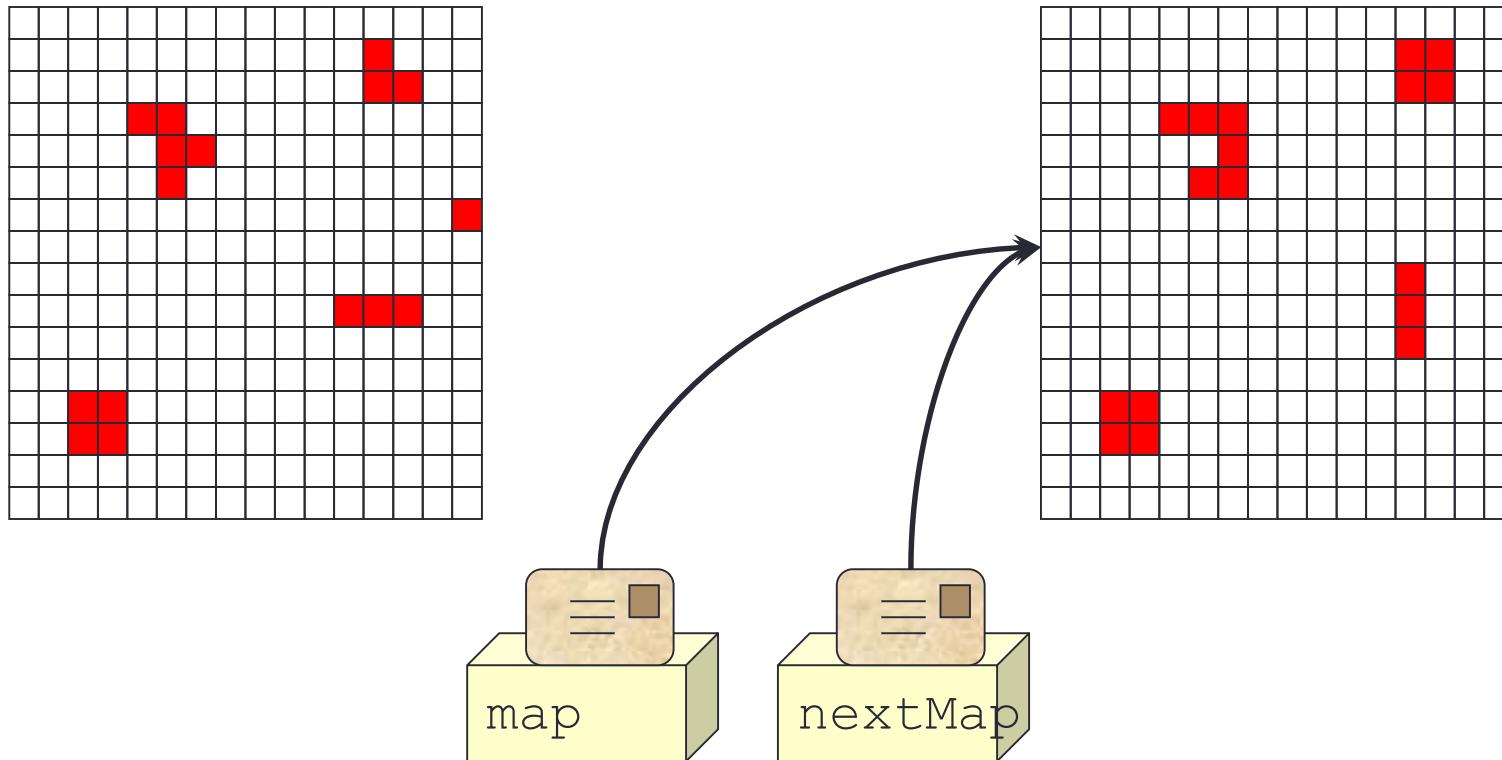
# A performance issue

- In each generation, the next-generation map is created



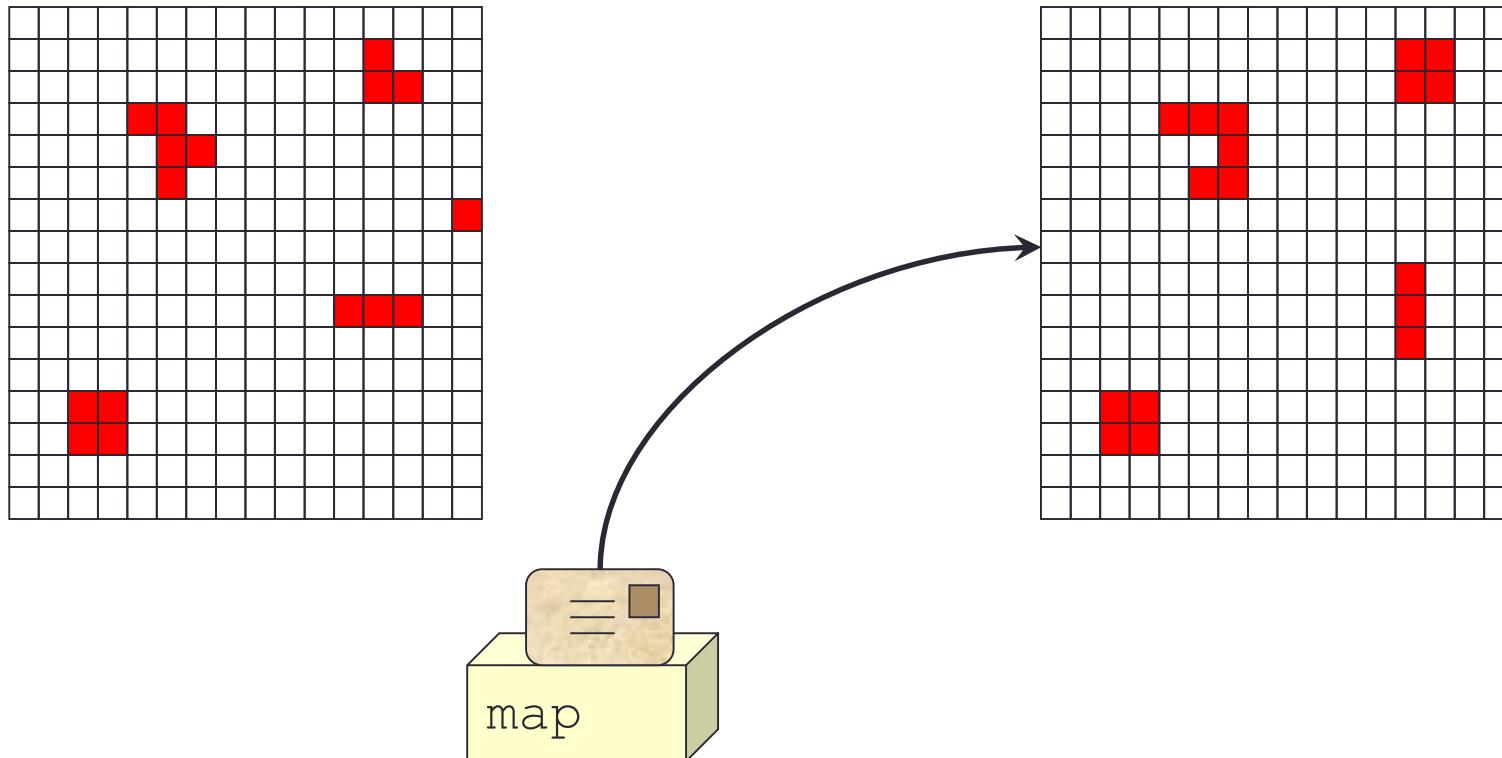
# A performance issue

- map is redirected to point to the new map



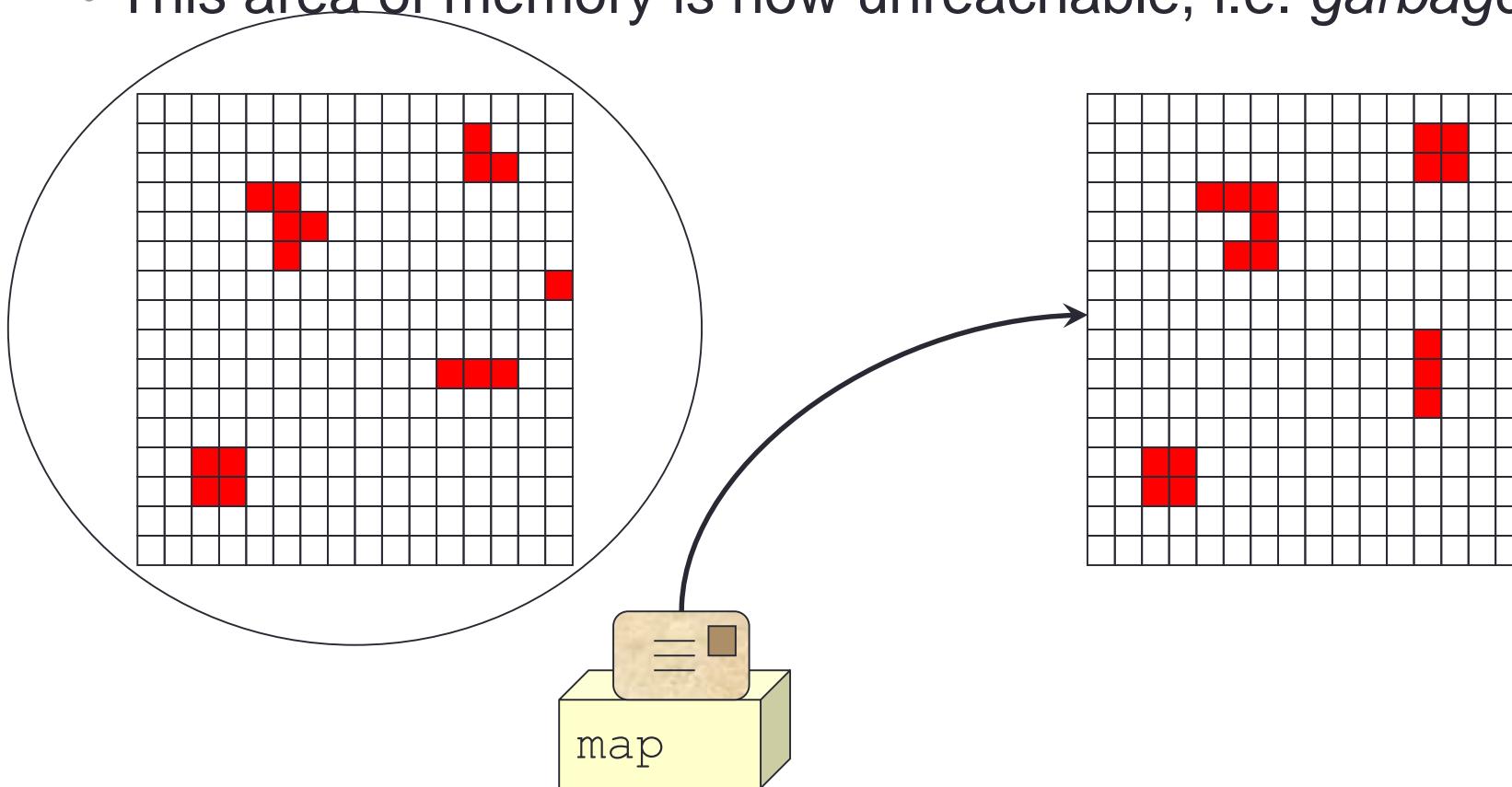
# A performance issue

- The local variable `nextMap` disappears



# A performance issue

- This area of memory is now unreachable, i.e. *garbage*



# Garbage collection

- Garbage objects can no longer play any role in the program's execution
  - The values they store cannot be accessed
- Java will automatically “collect” the garbage for you and recycle the space
  - But there is a performance penalty associated with this
- If the world is large and/or you are simulating many generations, the memory will rapidly fill up
  - The garbage collector will have to run often
  - This will interrupt the smooth “animation” of the program

# A solution

- We can fix this by keeping *two* “maps of the world” as ***instance variables*** and swapping them over at each generation

```
public class Life {  
    private boolean[][][] map, nextMap;  
    private int width, height;  
  
    // constructors & methods  
}
```

- The constructors will create a blank object for nextMap

# Changes to nextGeneration

- Delete the declaration of nextMap
  - We do not allocate new space in each generation
- Replace

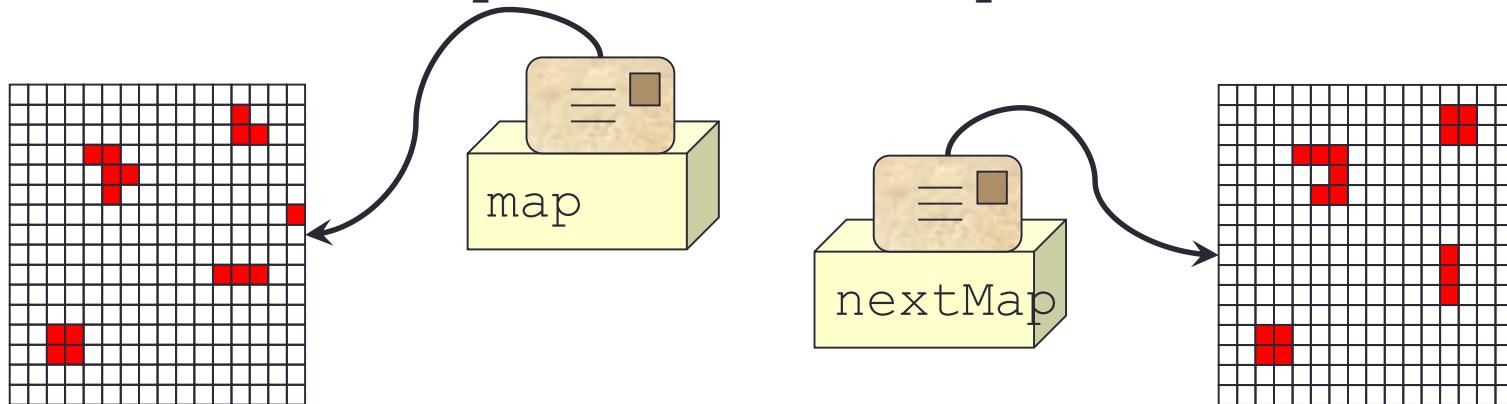
map = nextMap;

with

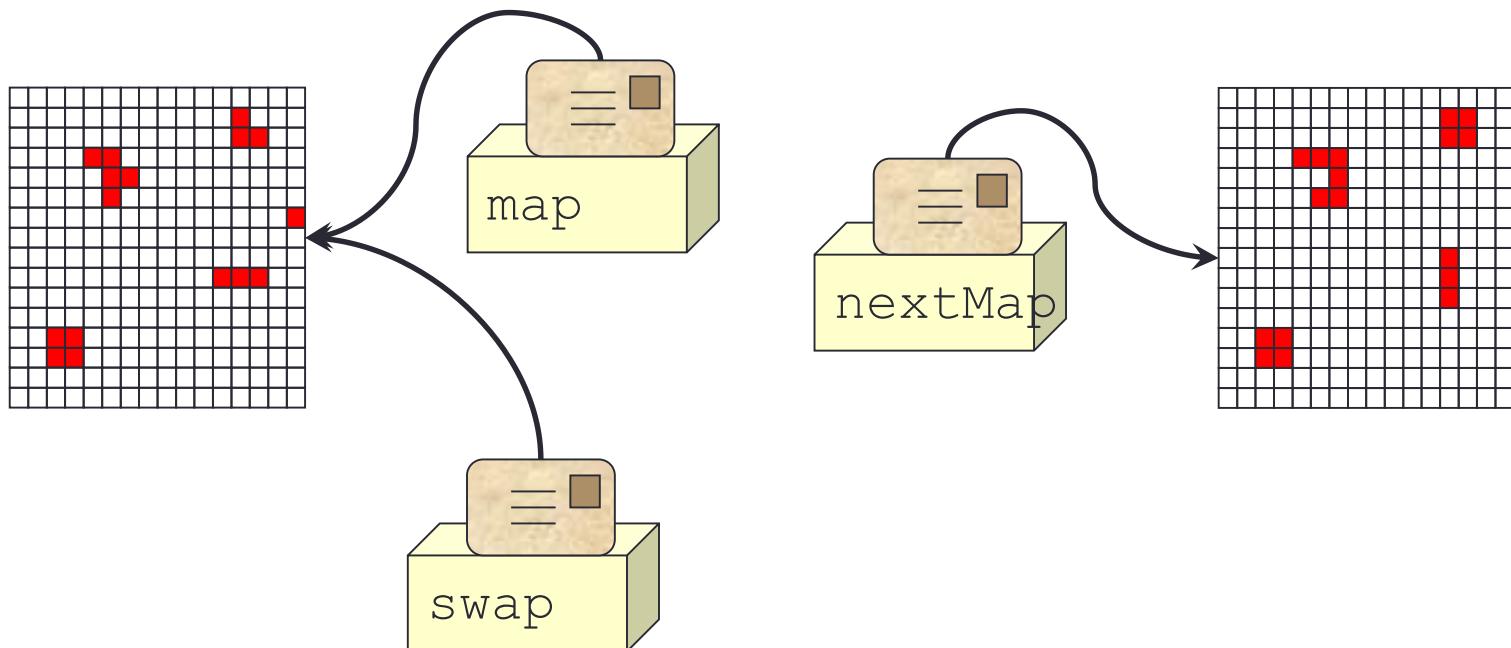
```
boolean[ ] [ ] swap = map;  
map = nextMap;  
nextMap = swap;
```

# How does this behave?

- The code uses `map` to fill in `nextMap`, then

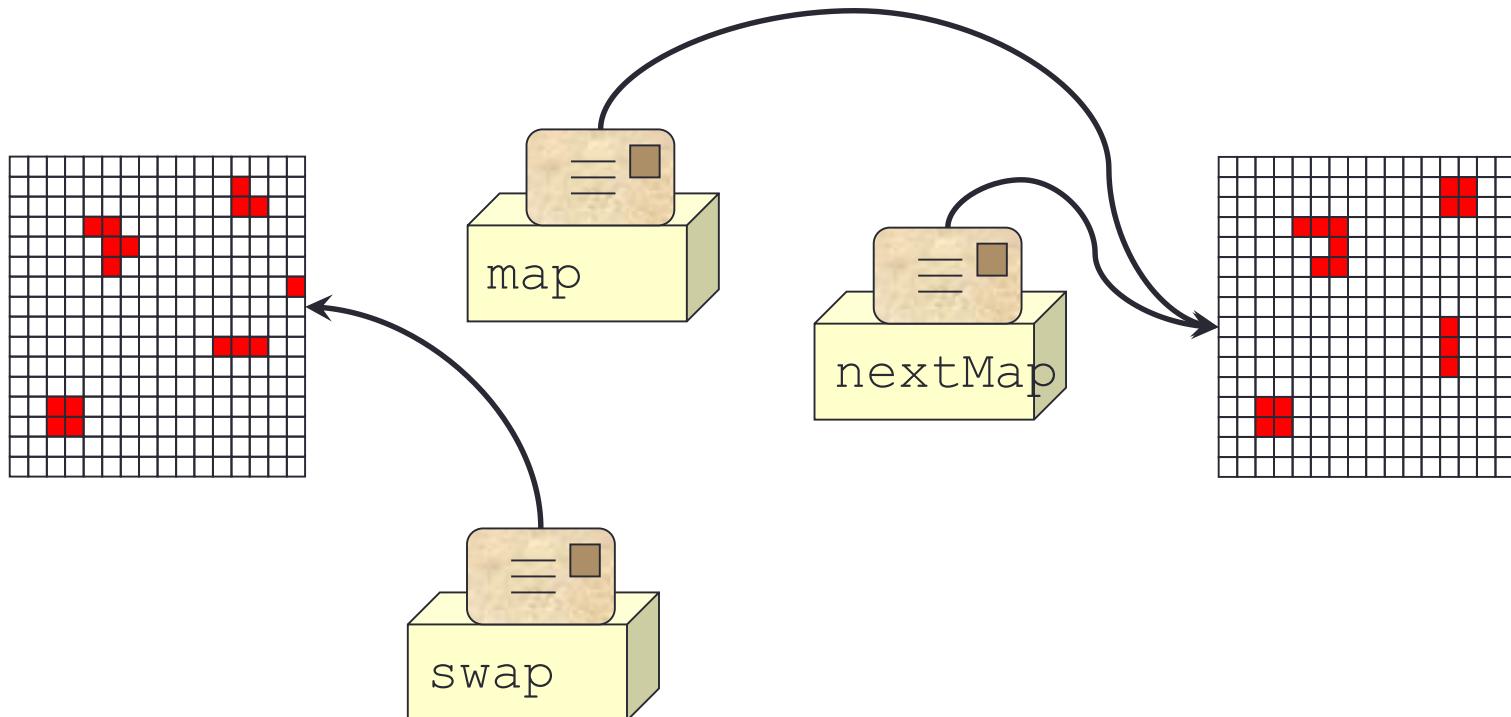


# How does this behave?



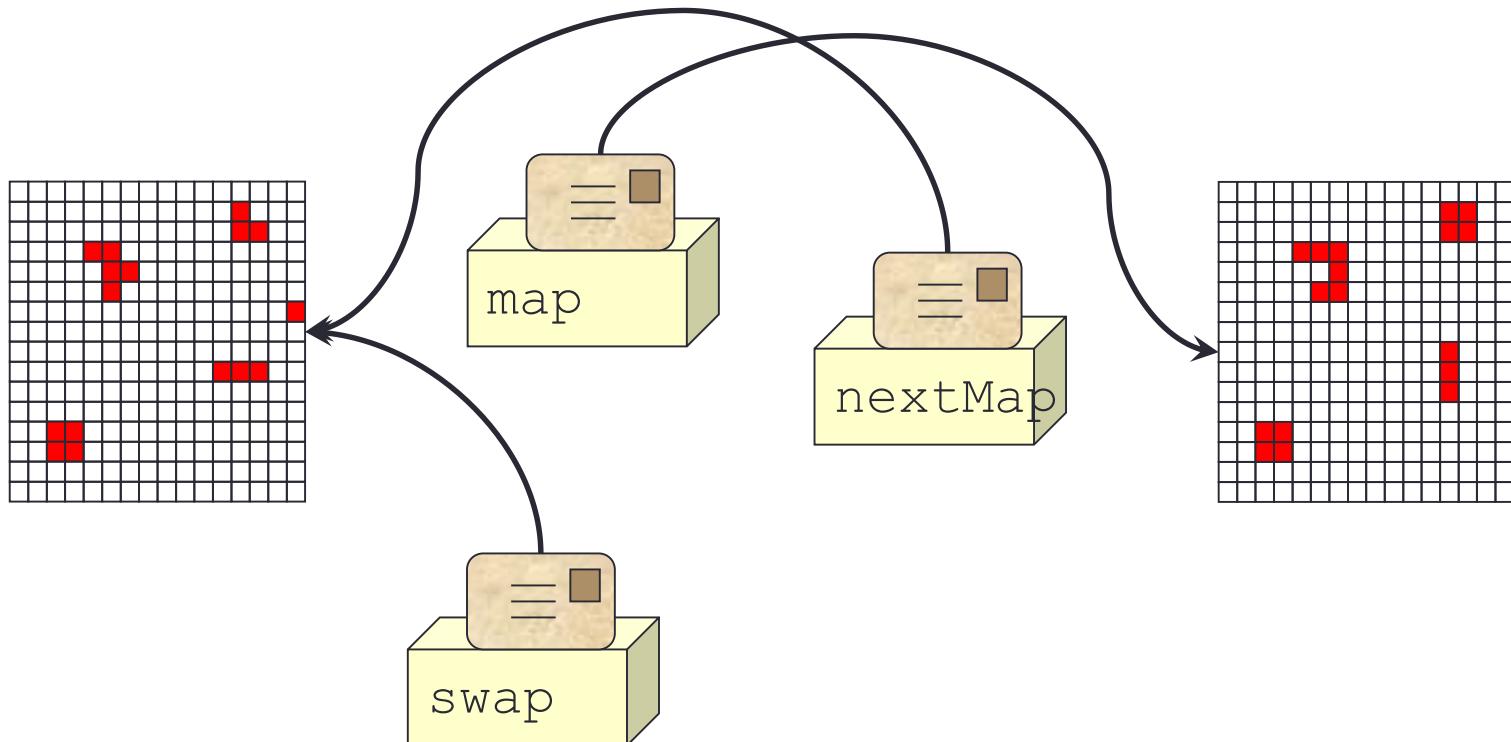
```
swap = map;
```

# How does this behave?



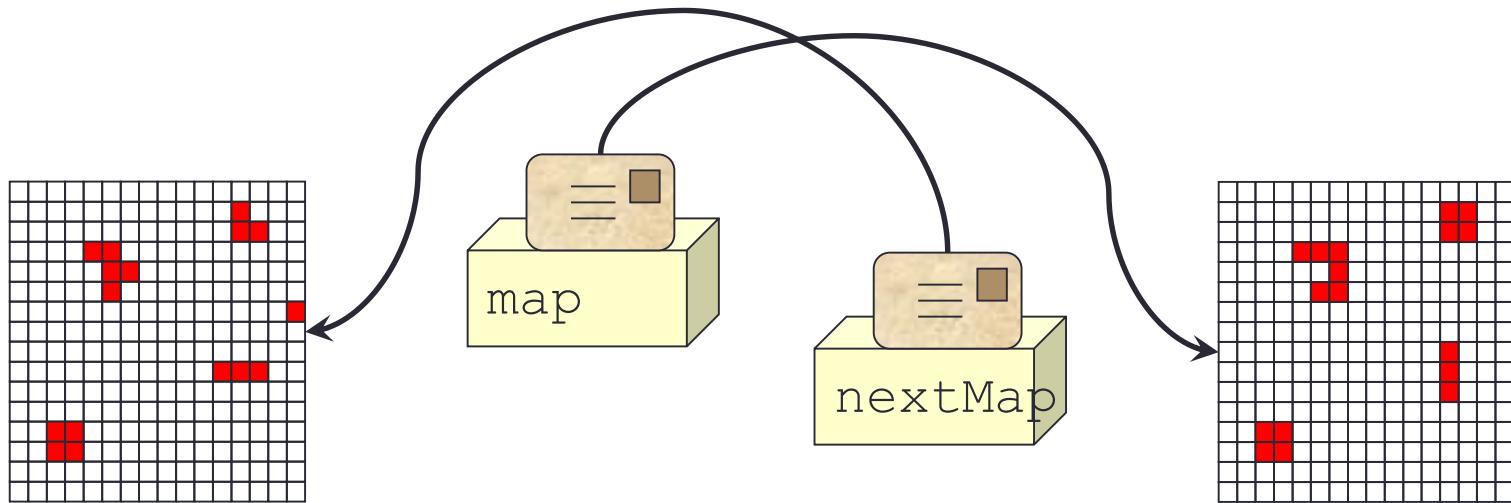
```
map = nextMap;
```

# How does this behave?



```
nextMap = swap;
```

# How does this behave?



swap disappears

# The new nextGeneration

```
public void nextGeneration() {  
    for (int i = 0; i < width; i++)  
        for (int j = 0; j < height; j++) {  
            int z = numNeighbours(i, j);  
            if (z == 2) nextMap[i][j] = map[i][j];  
            else         nextMap[i][j] = z == 3;  
        }  
    boolean[][] swap = map;  
    map = nextMap;  
    nextMap = swap;  
}
```

# Program design

- Recall that our program design called for three classes
  - Life
  - LifeViewer
  - SimpleCanvas
- The internal structure of the Life class has been written
- Now we must decide
  - How LifeViewer interacts with Life
  - The internal structure of LifeViewer

# Interaction

- LifeViewer needs Life to
  - Send the data for the current generation to be displayed
  - Update to the next generation when required
- This means that Life must have two public methods

```
public boolean[][] getMap()
```

- This returns the array representing the current map

```
public void nextGeneration()
```

- This updates the Life object to the next generation

# The extra method for Life

- An accessor method for the field `map`

```
public boolean[][] getMap() {  
    return map;  
}
```

# STOP ARGUING OVER THE BEST PROGRAMMING LANGUAGE

C is LOW-LEVEL

C++ is POWERFUL

Python is INTUITIVE

Rust is SAFE

Lua is EASY

Java

C# is LEGIBLE

# LifeViewer instance variables

- Each `LifeViewer` is responsible for one `Life` object, and it has one `SimpleCanvas` on which to draw
- The static variables define the “look” on the screen

```
public class LifeViewer {  
    private Life life;  
    private int width, height;  
    private SimpleCanvas sc;  
  
    private final static int CELL_SIZE = 4;  
    private final static Color BACK_COLOUR = Color.white;  
    private final static Color CELL_COLOUR = Color.red;  
    private final static Color GRID_COLOUR = Color.black;
```

# The LifeViewer constructor

- The constructor initialises the object's instance variables

```
public LifeViewer(Life life) {  
    this.life = life;  
    width = life.getMap().length;  
    height = life.getMap()[0].length;  
    sc = new SimpleCanvas("Life",  
        width * CELL_SIZE + 1,  
        height * CELL_SIZE + 1,  
        BACK_COLOUR);  
    display();  
}
```

# Code dissection

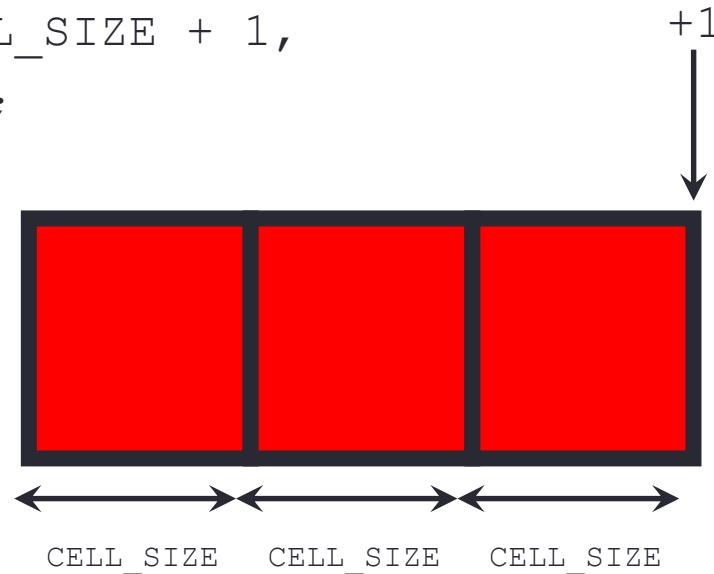
- The user must create the `Life` object and pass that in to the `LifeViewer` object
- `width` and `height` are maintained for convenience

```
public LifeViewer(Life life) {  
    this.life = life;  
    width = life.getMap().length;  
    height = life.getMap()[0].length;
```

# Code dissection

- The dimensions of the window are implicit from the number of cells and the size of each one
  - The +1 is for the right-hand and bottom borders

```
sc = new SimpleCanvas("The Game of Life",
                      width * CELL_SIZE + 1,
                      height * CELL_SIZE + 1,
                      BACK_COLOUR);
```



# Code dissection

- To display the world we need to
  - Get the current map from the Life object
  - Draw the cells, and then draw the grid

```
display();
```

- (It would be better to redraw only the cells that were changed in the last generation – how would we do that?)

```
private void display() {  
    drawCells();  
    drawGrid();  
}
```



# Code dissection

```
sc.drawRectangle(    i * CELL_SIZE,      j * CELL_SIZE,  
                  (i+1) * CELL_SIZE, (j+1) * CELL_SIZE,  
                  col);
```

- The  $[i][j]$  cell has  $i$  cells on its left and  $j$  cells above it
  - So across the screen, it starts at  $i * \text{CELL\_SIZE}$  and it finishes at  $(i+1) * \text{CELL\_SIZE}$
  - And similarly down the screen, in terms of  $j$

# Drawing the grid

- Draw the vertical lines, then draw the horizontal lines

```
private void drawGrid() {  
    for (int i = 0; i <= width; i++)  
        sc.drawLine(i * CELL_SIZE, 0,  
                    i * CELL_SIZE, height * CELL_SIZE,  
                    GRID_COLOUR);  
  
    for (int j = 0; j <= height; j++)  
        sc.drawLine(0, j * CELL_SIZE,  
                    width * CELL_SIZE, j * CELL_SIZE,  
                    GRID_COLOUR);  
}
```

**constant x coordinate  
gives a vertical line**

**j \* CELL\_SIZE**

**j \* CELL\_SIZE**

**constant y coordinate  
gives a horizontal line**

# Animate the world

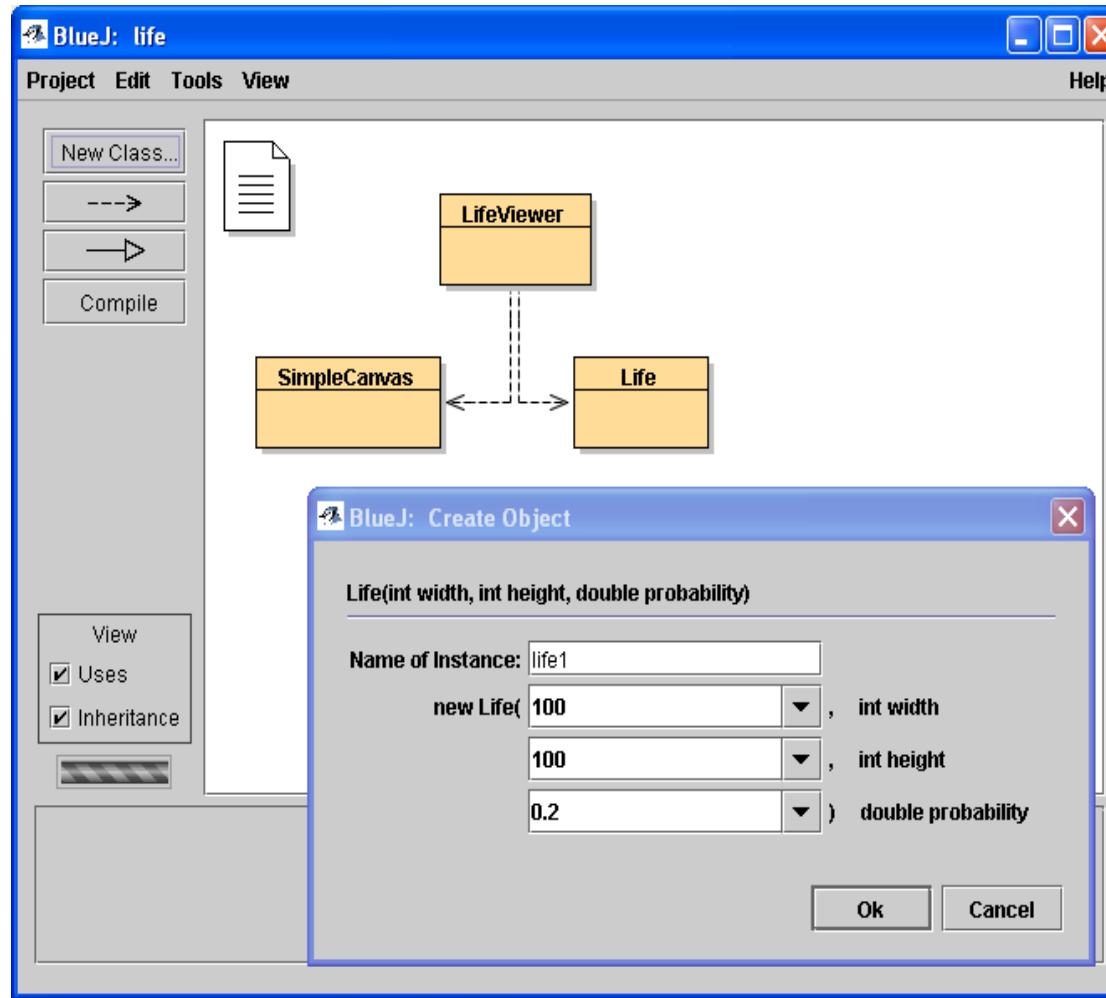
- In each generation, we advance the world, display the new state, and pause for effect

```
public void animate(int n) {  
    for (int i = 0; i < n; i++) {  
        life.nextGeneration();  
        display();  
        sc.wait(100);  
    }  
}
```

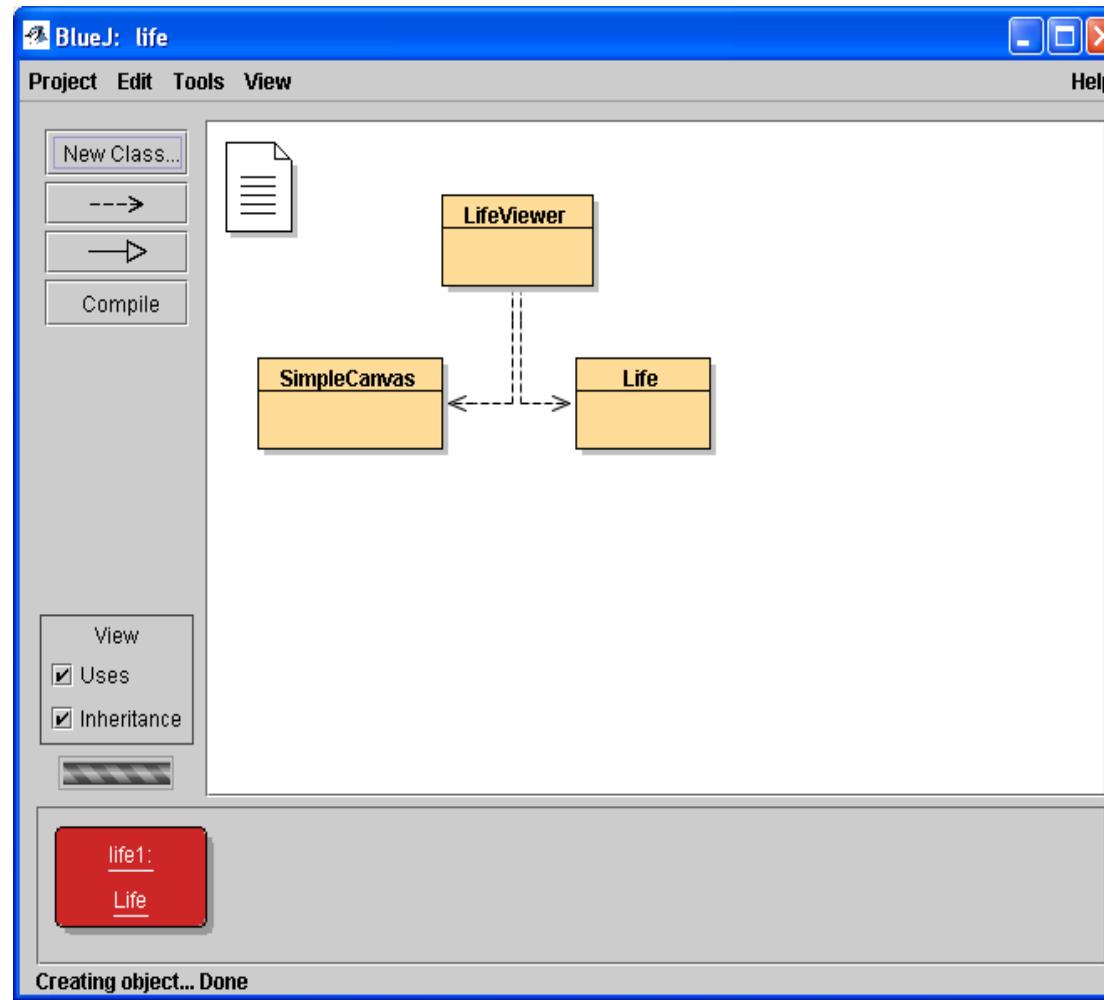
# SimpleCanvas

- The final component of the system is the class SimpleCanvas, that manages the interaction with the operating system
- The use of this class is described in Lecture 14, along with MouseListener and Color

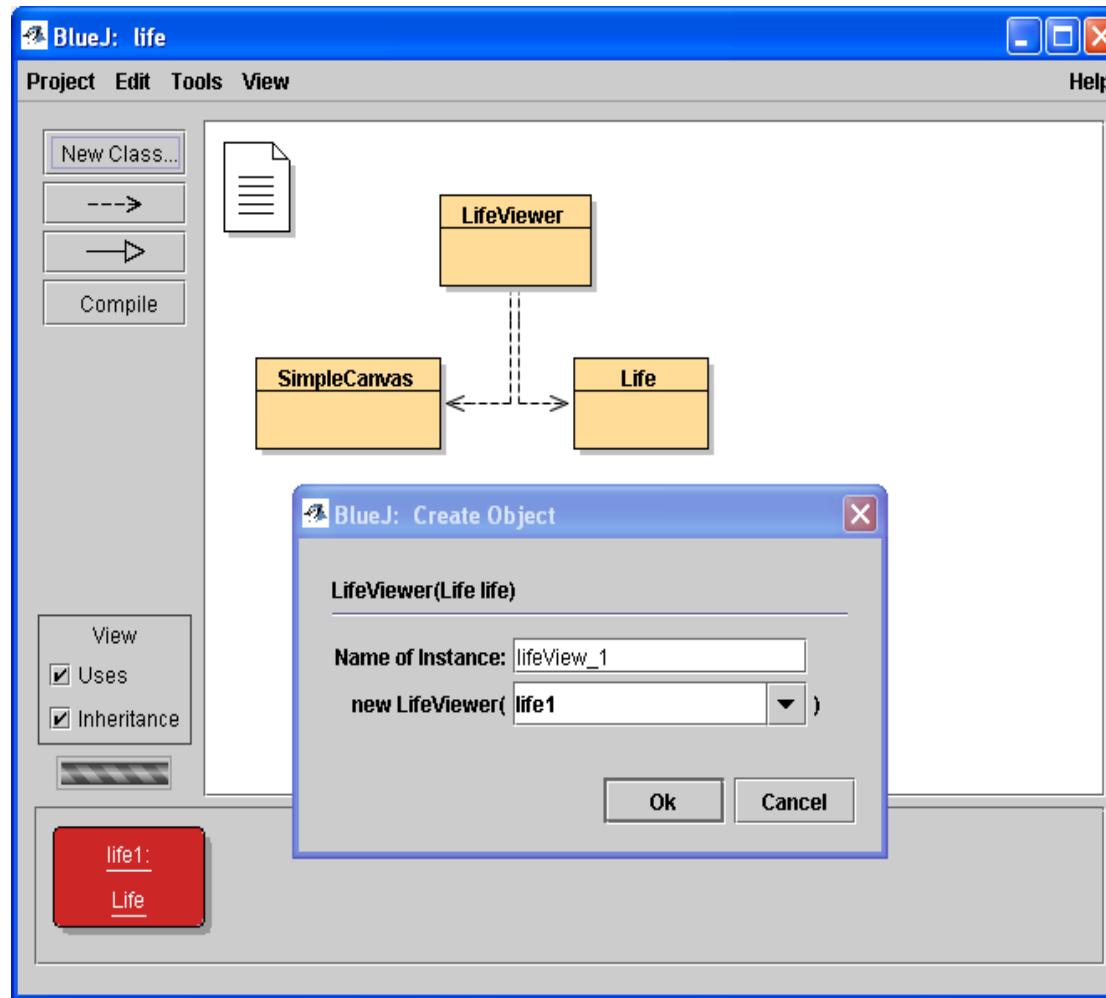
# Create a Life object



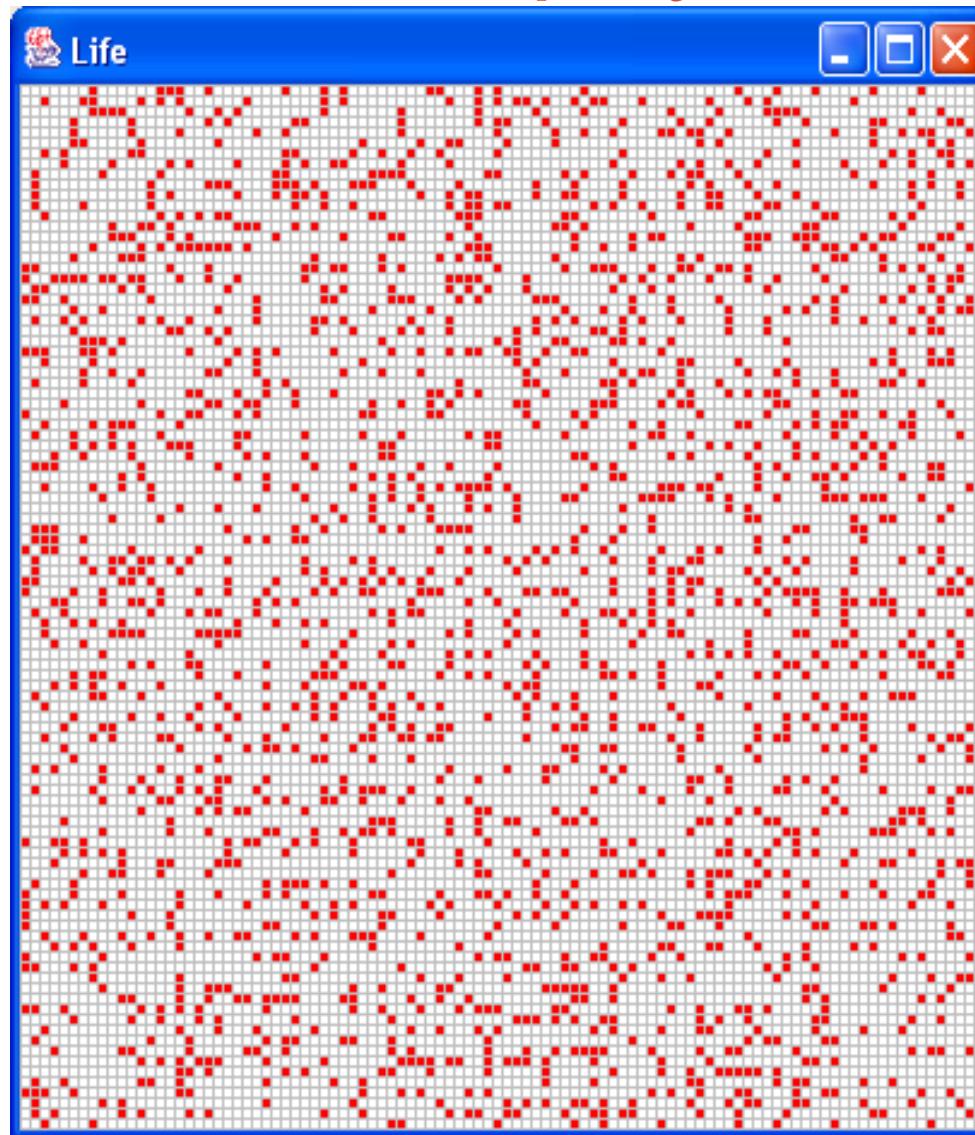
# It appears on the object bench



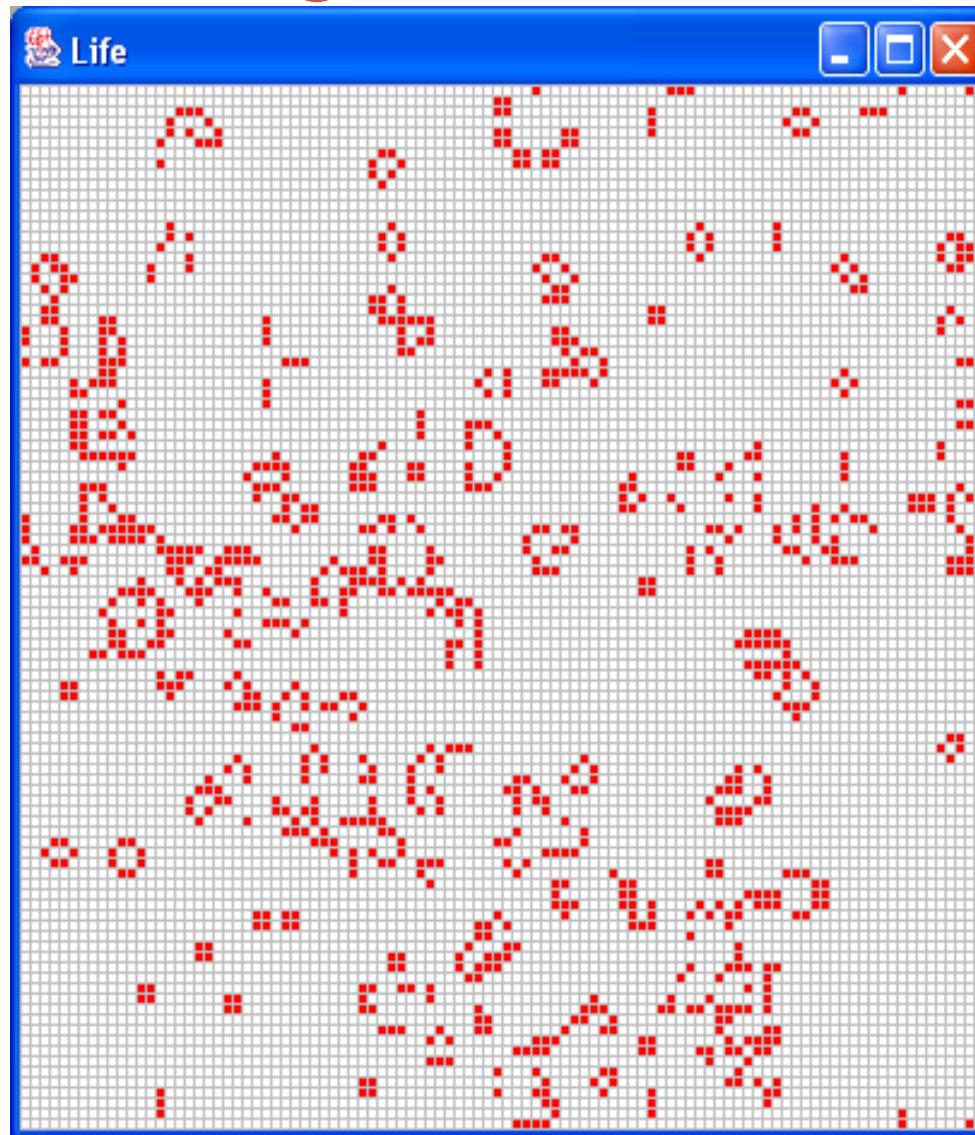
# Create a LifeViewer object



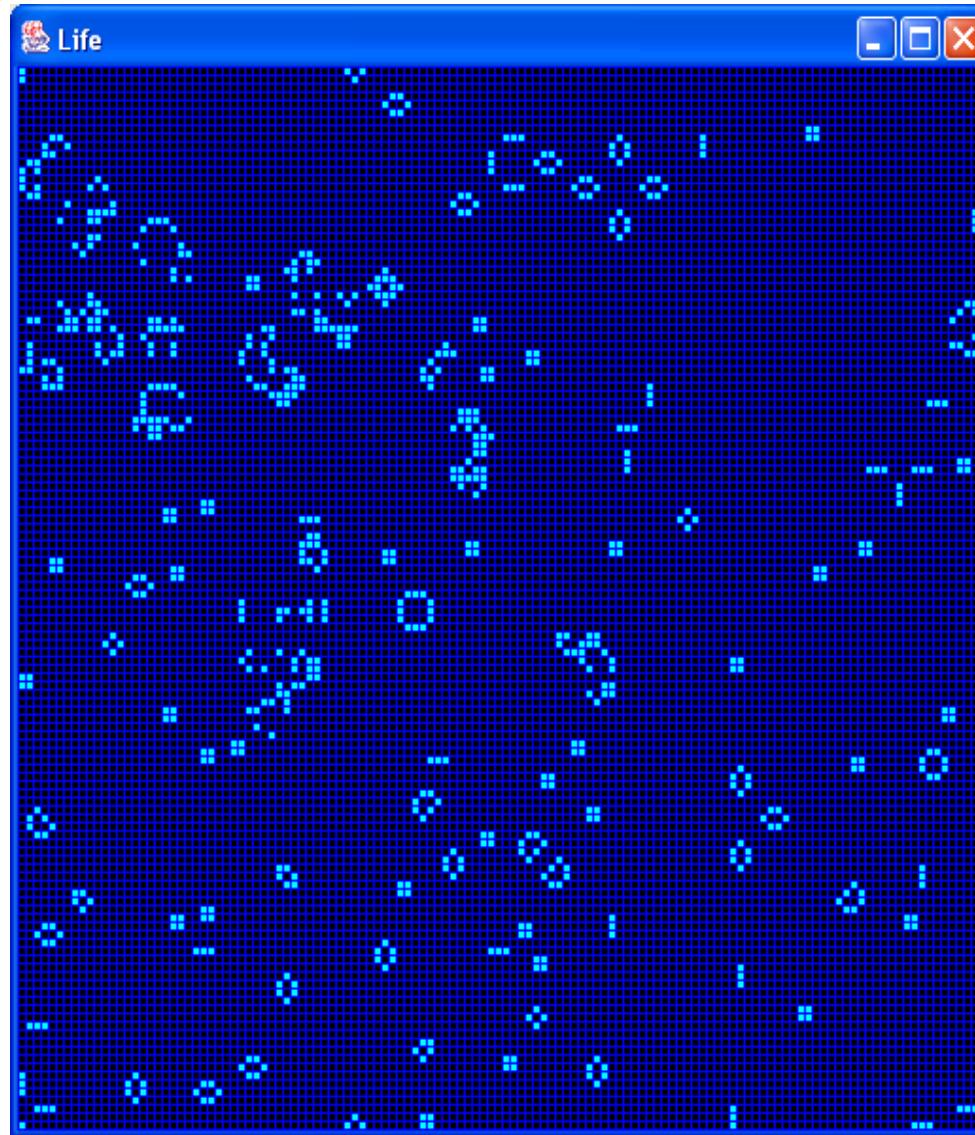
# Generation 0 is displayed



# Animate 100 generations



# Varying the colour scheme



# CITS1001

## 14. SIMPLECANVAS AND MOUSELISTENER – AN INTRO

---

# Lecture essentials

- A brief introduction to using the class SimpleCanvas to produce drawings and animations in Java
- You will also want to be familiar with some library classes
  - `java.awt.Color` – very simple
  - `java.awt.event.MouseListener` – fairly simple
  - `java.awt.Font` – fairly simple
- And maybe others too
  - Have a look!

# Introduction

- Some of the later material in CITS1001 produces visual imagery displayed on the computer screen
  - The Game of Life
  - The Fifteen Puzzle
- This is done with the class `SimpleCanvas`
  - Originally from the BlueJ team, modified for use in CITS1001
- Remember that in computer graphics,  
the convention is that the origin for coordinates  
is in the *top-left* corner of the window

# Creating a window

- SimpleCanvas has one complicated-looking constructor that takes four parameters
  - Don't worry about the body of the constructor!
  - You just need a title for the window, its width and height, and its background colour
- It also has a default constructor that sets these parameters for you
  - "SimpleCanvas", 400 x 400 pixels, white

```
SimpleCanvas sc = new SimpleCanvas();
```

- Note the use of `this` in the second constructor
  - The second constructor basically just calls the first

# Drawing stuff

- SimpleCanvas provides several methods that allow you to draw things in your window
- The main six are
  - drawLine, drawPoint, drawRectangle
  - drawDisc, drawCircle
  - drawString – **two versions**
- e.g. sc.drawLine(x1, y1, x2, y2, col)  
draws a line from the point x1, y1 to the point x2, y2  
in colour col on sc
- Other methods allow you to control the font, timing, etc.

# Using colour

- Using colour is very easy with `java.awt.Color`
- Mostly you will just want the static fields
  - e.g. `Color.RED`, `Color.BLUE`
  - Many other colours are available
  - See docs at <https://docs.oracle.com/javase/7/docs/api/java/awt/Color.html>
- You can also use the constructors to build new colours
  - e.g. `new Color(255, 0, 255)` is purple
  - The three arguments are red, green, and blue, each in 0–255
  - Trial and error is a good approach here...
- There are plenty of other methods too; have a look

# Some tips

- It takes a bit of practice getting stuff to appear exactly where you want it
- The Game of Life lecture gives some great examples of a grid-type display
- Remember that
  - Sketching things first on paper always helps
  - You can draw stuff on top of other stuff
  - You can draw stuff in loops
  - With for loops, the loop variable is often crucial
  - Trial and error is a good way to figure things out

# Using the mouse

- Once your class has created a window, it is very easy to connect it to the mouse
  - This is done most easily with the library class `MouseListener`
  - Every time the user makes a mouse action in your window, a method in your class is invoked
  - There are five changes required
- 
- (If you are feeling ambitious, you could also look at `MouseMotionListener`)

# 1. Import the required library classes

```
import java.awt.event.*;
```

Add this in the file containing your class

## 2. Tell Java that the class implements MouseListener

```
public class ClassName
```

should become

```
public class ClassName implements MouseListener
```

### 3. Associate MouseListener with your SimpleCanvas

```
sc.addMouseListener(this);
```

Add this in your constructor(s) after you initialise sc

## 4. Add these five methods to your class

```
public void mouseClicked (MouseEvent e) { }
```

```
public void mousePressed (MouseEvent e) { }
```

```
public void mouseReleased (MouseEvent e) { }
```

```
public void mouseEntered (MouseEvent e) { }
```

```
public void mouseExited (MouseEvent e) { }
```

- Whenever the user does something with the mouse in your window, the appropriate method is invoked
- The argument `e` holds info about what the user did
  - `e.getX()` and `e.getY()` give you the coordinates of the event

## 5. Implement some functionality

- This will write **Hello!** wherever the mouse is pressed

```
public void mousePressed (MouseEvent e) {  
    sc.drawString("Hello!", e.getX(), e.getY(),  
                 java.awt.Color.RED);  
}
```

- This will draw a green horizontal line of length 50 wherever the mouse is released

```
public void mouseReleased (MouseEvent e) {  
    sc.drawLine(e.getX(), e.getY(), e.getX() + 50, e.getY(),  
                java.awt.Color.GREEN);  
}
```

**constant y coordinate  
gives a horizontal line**

# javax.swing.SwingUtilities

- Another thing you might want to do is differentiate between different mouse buttons

```
static boolean isLeftMouseButton(MouseEvent e)
```

```
static boolean isMiddleMouseButton(MouseEvent e)
```

```
static boolean isRightMouseButton(MouseEvent e)
```

- There is a small example in SimpleMouseExample4

# Examples to play with

- The classes `SimpleMouseExample[1, 2, 3, 4]` use this material in simple ways
- The Game of Life lecture
- The Fifteen Puzzle lab

# CITS1001

## 15. DEFENSIVE PROGRAMMING

---

Part 1

# Lecture essentials

- Why program defensively?
- Encapsulation
- Access Restrictions
- Documentation
- Unchecked Exceptions
- Checked Exceptions
- Assertions



"No, defensive driving does not mean you hit the other guy first!"

# Why program defensively?

- Normally, your classes will form part of a larger system, so other programmers will use them and rely on them
- We use the term *client code* to mean code written by other programmers that uses your classes
- Obviously, your classes should be *correct*
- Also importantly, your classes should be *robust*; resistant to accidental (or non-accidental) misuse by other programmers
- You should aim to ensure that no errors in the final system can be attributed to the behaviour of *your* classes

# Correctness

- It may seem like a trite question, but what does it mean to ask if a code fragment is *correct*?
- It means that the code *behaves as expected*
  - These expectations may be in the values returned, or in the on-screen behaviour, or in the timing, or in the security aspects, or ...
  - Or of course some combination of these
- Where are these expectations stated?
  - Usually in the documentation for the code, or in some equivalent place
- Importantly, these expectations must be determined *independently of the code itself*
  - Usually before the code is written

# Encapsulation

- One of the most important features of object-oriented programming is that it facilitates *encapsulation*
- This means that a class describes both the data it uses, and the methods used to manipulate that data
- The external user sees *only* the public methods of the class, and interacts with objects belonging to that class purely by calling those methods
- This has several benefits
  - Users of the class can call the public methods without needing to understand their implementation or the representation of the data
  - Programmers can alter or improve the implementation of the class without affecting any client code
  - Use and implementation are *divorced*
- Abstraction again!

# Access restrictions

- Encapsulation is enforced by the correct use of access modifiers on instance variables and methods
  - `public`, `private`, `<default>`, and `protected`
- If you omit the access modifier, you get the default, sometimes known as “package”
- The latter two modifiers are relevant only for multi-package programs that use inheritance, so at the moment we need consider only `public` and `private`

# public and private

- If an instance variable is public
  - Any object can access it directly
  - Any object can alter it directly
- If an instance variable is private
  - Only objects that belong to *the same class* can access and alter it
- If a method is public
  - Any object can call that method
- If a method is private
  - Only objects that belong to *the same class* can call it
- Note that privacy is a per-class attribute, not per-object

# Public methods

- The *public interface* of a class is its list of public methods, which details all of the services that the class provides
- Once a class is released (e.g. as part of a library), it is difficult to change its public interface without causing problems for client code
- Partly for this reason, classes should make *as few methods public as possible*
  - Limit them to just the methods needed for the class to perform its stated function
- Public methods must be
  - Precisely documented
  - Robust to incorrect input and accidental misuse

# Public variables

- Normally instance variables should **not** be public
- If client code can alter the values of instance variables, the benefits of encapsulation are lost
- If client access to instance variables is desirable, it should be provided by accessor and/or *mutator* methods (getters and setters)
- There are two important advantages to this
  - Permits change of representation
  - Maintains object integrity

# A simple example

```
class MyDate {  
    public int day;  
    public String month;  
    public int year;  
  
    ...  
}
```

```
MyDate md = new MyDate();  
md.day = 31;  
md.month = "Feb";
```

md is corrupt and could  
cause problems elsewhere  
in the system

# Use mutators instead

```
public void setDay(int day) {  
    // Check that day is valid for this.month  
    // before setting the variables  
}
```

```
public int getDay() {  
    return day;  
}
```

- Setter methods act as “gatekeepers” to protect the integrity of objects
- Setters reject values that would create a corrupt object
- Getters return a value for client code to use, but do not allow the object itself to be changed

# Avoid accessors returning objects

```
private int[] scores;  
  
public int[] getScores() {  
    return scores; ←  
}  
}
```

**returns the address of  
the internal array object**

- Getters that return objects might still create problems via aliasing
    - This can be avoided by copying the object
    - Or sometimes it can be avoided by using e.g. indexing

```
public int getScore(int k) {  
    if(0 <= k && k < scores.length) return scores[k];  
    else // what goes here?  
}
```

# Public variables

- The principal exception to this rule is when a public variable is also `final`
- Such variables can't be assigned to, so they can't be corrupted
- Using `final` variables in place of so-called “magic numbers” promotes both readability and maintainability
- In many contexts, `final` variables are *intended* to be used as widely as possible

# Take a break



# An aside: magic numbers

- *Magic numbers* in programming are “values with unexplained meaning or multiple occurrences which could (preferably) be replaced with named constants”
- e.g. if you see the number 55 in someone’s code, what is it?
  - Someone’s age? Part of their address? Their mark in CITS1001?
  - `int mark1001 = 55;` will lead to much clearer code
- e.g. if the size of a window is 400, you may need to refer to this value many times
  - *Do not* repeat the number; use instead something like  
`final int WINDOW_SIZE = 400;`
  - Also replace e.g. 200 with `WINDOW_SIZE / 2` if you need to refer to the centre of the window

# Avoiding magic numbers

- Promotes readability
  - Names are easier to understand than numbers
- Promotes maintainability
  - A single assignment is easier to change than repeated numbers
- Even helps you to catch typos!
  - The system cannot tell you that 4000 is wrong, but it can tell you that WINDOW\_SIZE is wrong
- Other advantages are listed at  
[https://en.wikipedia.org/wiki/Magic\\_number\\_\(programming\)#Unnamed\\_numerical\\_constants](https://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants)

# Magic numbers example

- There is a great little example of poor practice in this regard in the *SimpleMouseExample3* project on the LMS

```
// draws a purple cross centred at x, y  
sc.drawLine(x-15, y-15, x+15, y+15, purple);  
sc.drawLine(x-15, y+15, x+15, y-15, purple);
```

- All of these 15s should always be the same
  - Better to name the size of the cross, in case it ever changes

```
int d = 15;  
sc.drawLine(x-d, y-d, x+d, y+d, purple);  
sc.drawLine(x-d, y+d, x+d, y-d, purple);
```

# Documentation

- For large systems, documentation should be developed *at the same time, and in the same place*, as the code
  - A common occurrence is that documentation is delayed until the end of the project – and it doesn't happen!
- The documentation is a *contract* between the programmer and the users of a class
  - It describes what the class *is meant to do*
- Java provides facilities to help with this, by allowing code comments to be automatically turned into documentation
  - The Javadoc feature

# Javadoc

- “Normal” Java comments use two types of syntax
  - `//` comment to the end of this line
  - `/*` comment to the closing “tag” `*/`
- Javadoc comments occur between different tags
  - `/**` Javadoc comment to the closing “tag” `*/`
  - Note the `/**` at the front!
- These can be automatically processed to generate HTML documentation, used to precisely describe the behaviour of the class and its methods
  - The Java API itself is generated from source code comments in Javadoc

# Javadoc comment style

- Place comments directly before the relevant class, instance variable, constructor, or method
- Critically important for public entities, as it forms part of the contract
  - Sometimes also useful for private entities
- Comments can be written in HTML and enhanced with special Javadoc tags
  - e.g. @author, @version, @param, @return, @throws, @see

# Javadoc example

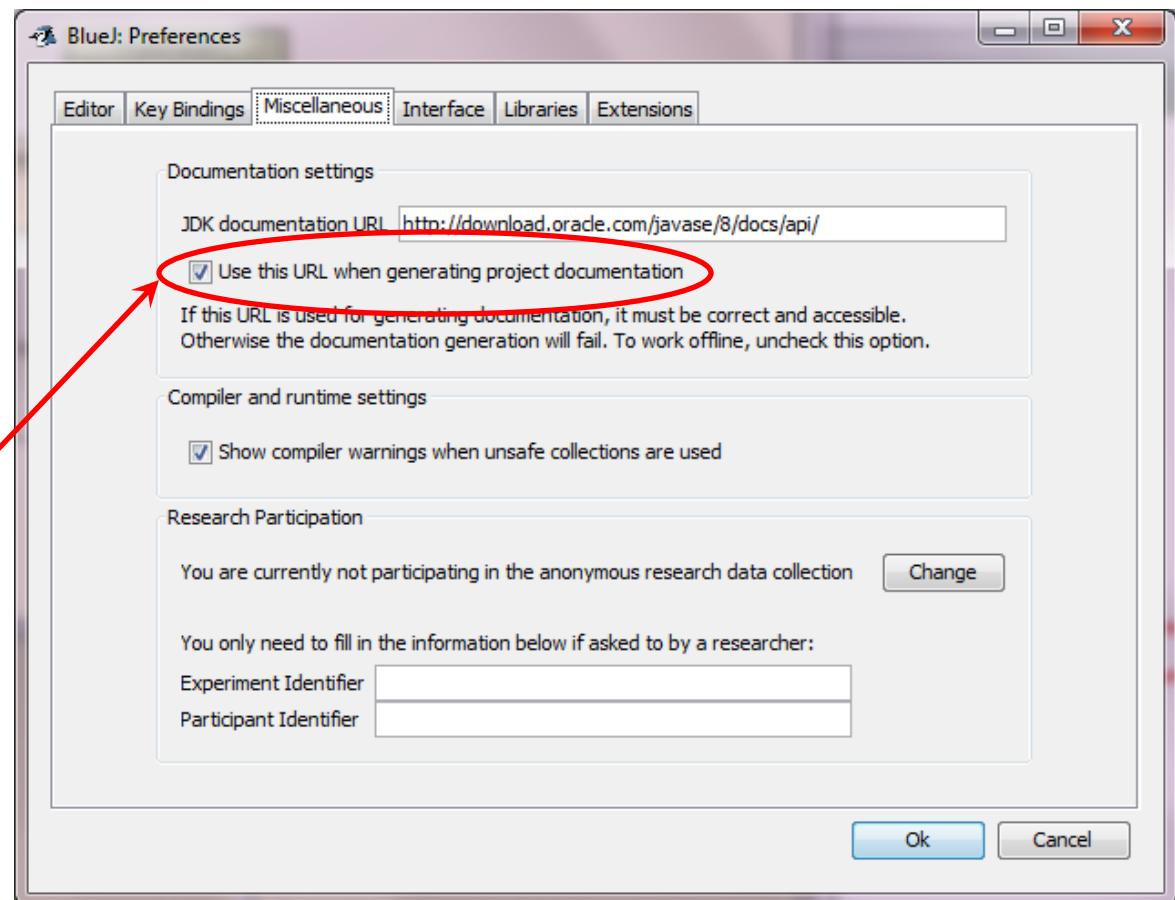
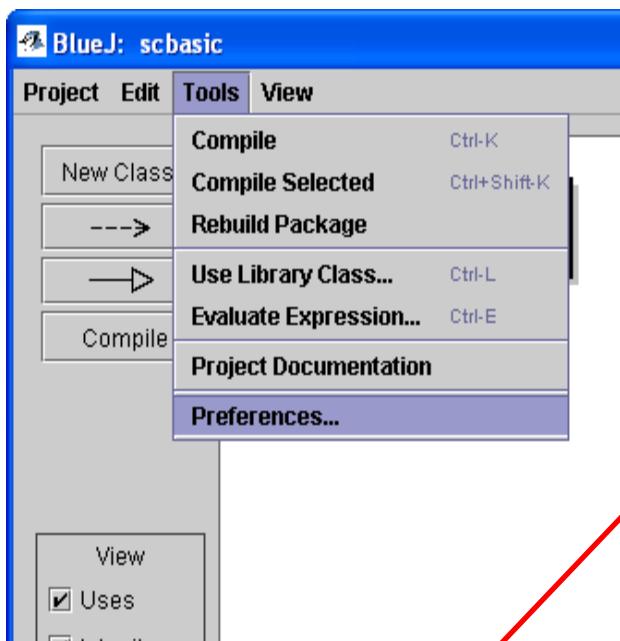
The screenshot shows a Java code editor window titled "SimpleCanvas". The menu bar includes "Class", "Edit", "Tools", and "Options". The toolbar contains "Compile", "Undo", "Cut", "Copy", "Paste", and "Close" buttons. A dropdown menu labeled "Implementation" is open. The code area contains the following Java code with Javadoc comments:

```
private JFrame frame;
private CanvasPane canvas;
private Graphics2D graphic;
private Image canvasImage;
private boolean autoRepaint;

/**
 * Creates and displays a SimpleCanvas of the specified size
 * with a white background. The client specifies whether repainting
 * after a drawing command should be manual or automatic.
 *
 * @param title title for the window
 * @param width the desired width of the SimpleCanvas
 * @param height the desired height of the SimpleCanvas
 * @param autoRepaint true for automatic repainting
 */
public SimpleCanvas(String title, int width, int height, boolean autoRepaint)
{
    frame = new JFrame();
    canvas = new CanvasPane();
    frame.setContentPane(canvas);
}
```

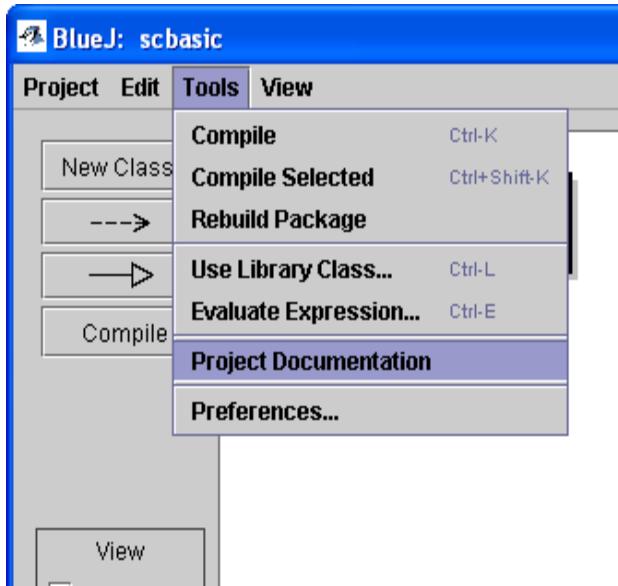
The status bar at the bottom says "Class compiled - no syntax errors" and has a "saved" button.

# Adjust BlueJ's preferences



Uncheck this if offline, or  
it will try to link to Sun's  
Java documentation

# Generate documentation



A screenshot of Microsoft Internet Explorer showing a JavaDoc-style documentation page. The title bar says "SimpleCanvas (scbasic) - Microsoft Internet Explorer". The address bar shows "C:\Documents and Settings\administrator.PC-214\Desktop\lectures03-2\code\scbasic\doc\index.html".  
**All Classes**  
[SimpleCanvas](#)  
  
Creates and displays a SimpleCanvas with a white background and with automatic repainting after drawing commands.  
  
**Parameters:**  
title - title for the window  
width - the desired width of the SimpleCanvas  
height - the desired height of the SimpleCanvas  
  
**SimpleCanvas**  

```
public SimpleCanvas(java.lang.String title,  
                   int width,  
                   int height,  
                   boolean autoRepaint)
```

  
Creates and displays a SimpleCanvas of the specified size with a white background. The client specifies whether repainting after a drawing command should be manual or automatic.  
  
**Parameters:**  
title - title for the window  
width - the desired width of the SimpleCanvas  
height - the desired height of the SimpleCanvas  
autoRepaint - true for automatic repainting  
  
**Method Detail**

# To be continued..

- Part 2 will cover next week

# CITS1001

## 15. DEFENSIVE PROGRAMMING

---

Part 2

# Lecture essentials

- Why program defensively?
- Encapsulation
- Access Restrictions
- Documentation
- Unchecked Exceptions
- Checked Exceptions
- Assertions



"No, defensive driving does not mean you hit the other guy first!"

# Dealing with errors

- Even if your classes are well-protected, errors will occur
  - *All* non-trivial software contains errors!
- We consider three types of error
  - Client code attempts to use your methods incorrectly, by passing incorrect parameter values
  - Your code cannot perform the services it is meant to, due to circumstances outside your control (e.g. an Internet site is unavailable)
  - Your own code behaves incorrectly and/or your objects become corrupted
- To handle these problems, Java provides
  - *Unchecked exceptions*
  - *Checked exceptions*
  - *Assertions*

# Invalid parameters

- `s.charAt(k)` returns the character at position `k` in String `s`
- Valid values for `k` are 0 up to `s.length() - 1`
- What happens if e.g. `s.charAt(-1)` is called?

# The method “throws” an exception

- If a parameter is invalid, the method cannot do anything sensible with the request
  - It creates an object from an `Exception` class and “throws” it
- If an `Exception` object is thrown, the runtime environment immediately tries to deal with it
  - If it is an *unchecked* exception, the system halts with an error message
  - If it is a *checked* exception, the system tries to find some object in the program able to deal with it
- The method `charAt` throws a `StringIndexOutOfBoundsException`
  - This is unchecked and hence causes the program to cease execution (i.e. to crash!)

# Throw your own exceptions

- Your own methods and/or constructors can throw exceptions, if client code calls them incorrectly
- This is how your code can enforce rules about how methods should be used
- For example, we can insist that the `deposit` method from the `TicketMachine` class is called with a positive value for the argument `amount`
- The general mechanism is to check the parameters, and if they are invalid in some way to then
  - Create an object from the class `IllegalArgumentException`
  - Throw that object

# Throw your own

```
public void deposit(int amount) {  
    if (amount > 0) balance += amount;  
    else throw new IllegalArgumentException(  
        "Deposited amount " +  
        amount + " was not positive");  
}
```

- If the amount is negative, *create* the object and *throw* it
- The constructor for `IllegalArgumentException` takes a `String` argument which is an error message that is presented to the user
- Throwing an exception is also often used by constructors to prohibit the construction of invalid objects

# Why throw unchecked exception?

- If the program is going crash, what is the use of throwing an unchecked exception?
- Throwing an exception is a standardized way to deal with errors.
- To provide informative feedback.
- You want to crash right where the problem happened and correct it rather than proceeding and waiting for things to go wrong later in the program – making it difficult to debug.

# “Predictable” errors

- Unchecked exceptions terminate program execution, and are used when the client code is seriously wrong
- Other error situations do not necessarily mean that the client code is incorrect, but reflect either a transient, predictable, or correctable mistake
- This is particularly common when handling end-user input, or when dealing with the operating system
- e.g. printers may be out of paper, disks may be full, web-sites may be inaccessible, filenames or URLs might be mistyped, etc.

# Checked exceptions

- Methods prone to such errors may elect to throw *checked exceptions*, rather than unchecked exceptions
  - With a checked exception, the program tries to recover without crashing
- Using checked exceptions is more complicated than using unchecked exceptions in two ways
  - The method is **required** to declare that it might throw a checked exception, and
  - All client code using that method is **required** to provide code that will be run if it does throw an exception

# The client perspective

- Many Java library classes declare that they *might* throw a checked exception

```
public FileReader(File file) throws FileNotFoundException
```

Creates a new FileReader, given the File to read from.

**Parameters:**

file - the File to read from

**Throws:**

FileNotFoundException - if the file does not exist,  
if it is a directory rather than a regular file, or  
for some other reason it cannot be opened for reading

# try and catch

- If code uses a method that might throw a checked exception, then it *must* enclose it in a try/catch block

```
try
{
    FileReader fr = new FileReader("lect.ppt");
    // code for when everything is OK
}
catch (java.io.FileNotFoundException e)
{
    // code for when things go wrong
}
```

- Try to open and process this file
- But be prepared to *catch* an exception if necessary

# Operation of try/catch

- Logically, try/catch operates a lot like if/else
- If everything goes smoothly
  - The code in the `try` block is executed, and
  - The code in the `catch` block is skipped
- If *any* of the statements in the `try` block causes an exception to be thrown
  - Execution immediately jumps to the `catch` block, which tries to recover from the problem
- What can the `catch` block do?
  - For human users: report the error and ask the user to change their request, or retype their password, or ...
  - In all cases: provide some feedback as to the likely cause of the error and how it may be overcome, even if ultimately it just causes execution to cease

# Multiple catch blocks

- Sometimes the `try` block might call several methods that can throw different checked exceptions
  - Or even one method that can throw several different checked exceptions...
- In this case multiple catch blocks are required
  - One to deal with each possible checked exception, e.g.

```
try {  
    // code for when everything is OK, but which  
    // might throw multiple checked exceptions  
}  
catch (java.io.FileNotFoundException e) {  
    // code for when things go wrong  
}  
catch (java.io.URISyntaxException e) {  
    // code for when things go wrong  
}
```

# The programmer perspective

- If you write a method that throws a checked exception, this must be declared in the source code, where you must specify the *type* of exception that might be thrown

```
public void printFile(String fileName) throws  
    java.io.FileNotFoundException {  
    // Code that attempts to print the file  
}
```

- If your method declares that it might throw a checked exception, the compiler will *force* any client code that uses it to enclose it in a try/catch block
- This explicitly makes the client code responsible for these situations
- Look at `FileNotFoundException` for a very simple example

# Using and testing exceptions

```
@Test(expected = IllegalArgumentException.class)
public void testIllegalDeposit() {
    deposit(-20);
}
```

- Java provides many exception classes that cover most common possibilities
- Exceptions are simply objects in a Java program, so you can write your own classes of exceptions if desired

# Checked or Unchecked?

- **Unchecked Exceptions**
  - Any method can throw them without declaring the possibility
  - No need for client code to use `try/catch`
  - Causes execution to cease
  - Used for fatal errors that are unexpected and that are unlikely to be recoverable
- **Checked Exceptions**
  - Methods must declare that they might throw them
  - Client code must use `try/catch`
  - Causes control flow to move to the `catch` block
  - Used for situations that are not entirely unexpected and from which clients may be able to recover
  - Use only if you think the client code might be able to do something about the problem

# Assertions

- Assertions are a debugging mechanism to use when you are developing complicated code
- At any point in your code, add a statement of the form

```
assert <boolean-condition> : <string>;
```

- When the assertion is executed, the boolean condition is evaluated
  - If it is true, execution continues
  - If it is false, execution is halted with an (unchecked) `AssertionError`, and the message string is printed

# Why use assertions?

- Assertions are used to help locate *logic errors*
- As you construct a complicated piece of code, mentally you should have a picture of what values a given variable *should* or *could* contain
- Use assertions to make this picture explicit, and to have the system check it for you during execution
- Otherwise an error might only become apparent a long time after the code that actually caused it, which makes it much harder to track down

```
if(condition) f();
```



```
if(condition == true) f();
```



```
if(make_pair(condition,!condition) ==  
    make_pair(true,false)) f();
```



```
if(vector<int>(int(condition)).size() != 0) f();
```



```
condition && f();
```



```
assert(condition);  
f();
```

# THE END

- One more lecture to go
- Then revision starts

# CITS1001

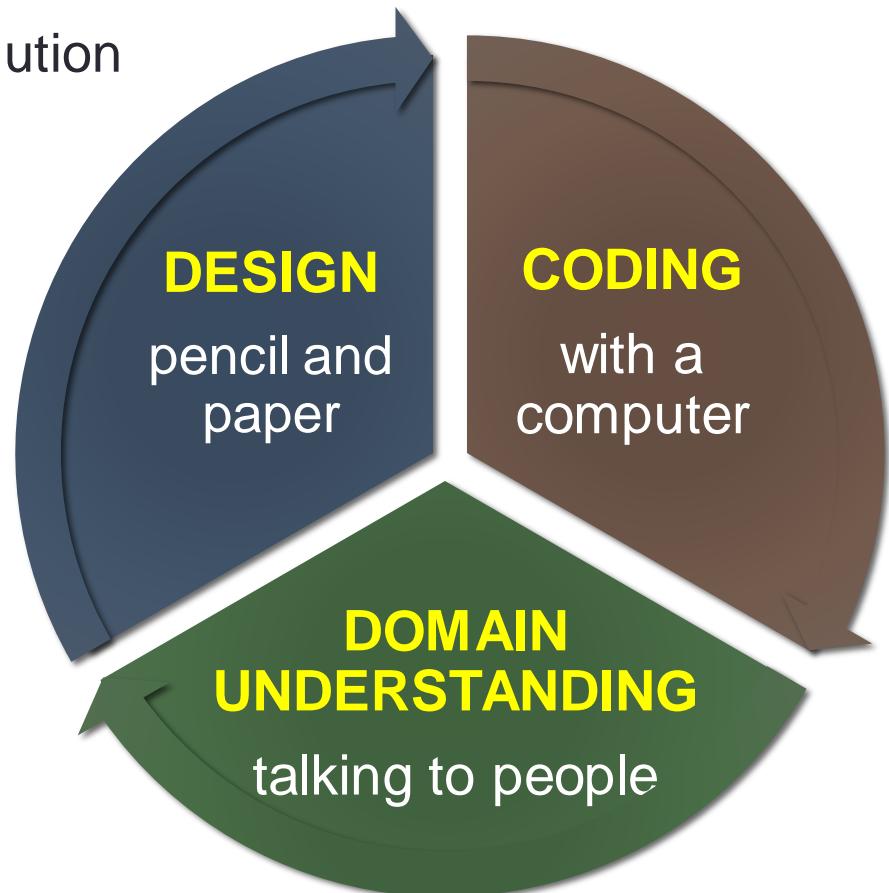
## 16. ELECTION NIGHT – A FINAL CASE STUDY

---

*Objects First With Java,*  
Chapter 9

# Lecture essentials

- Software development
  - Understand the problem
  - Understand and structure the solution
  - Implement and test the solution
  - And go around again
- Testing
  - Identifying the presence of bugs
- Debugging
  - Locating the bugs



# Preferential voting in the Australian House of Reps

- Each electorate has
  - $M$  candidates (we will use five candidates  $ABCDE$  in our examples)
  - $N$  formal votes
  - One winner, who needs  $N/2+1$  votes to triumph
- Each voter returns a list of preferences for all candidates
  - e.g.  $CAEDB$
  - First choice  $C$ , second choice  $A$ , etc.
  - But not  $CAE$  or  $CAEDA$  or  $CAXEDB$  or ...
- [http://www.aec.gov.au/Voting/counting/hor\\_count.htm](http://www.aec.gov.au/Voting/counting/hor_count.htm)

# The count

- Discard informal votes
- Allocate each formal vote to its first-choice candidate
- Then in each round of the count
  - If the candidate currently in the lead has enough votes:
    - Declare them the winner
  - Otherwise:
    - Identify the candidate  $X$  with the fewest votes
    - Redistribute each of  $X$ 's votes to its highest-ranked surviving candidate
    - Eliminate  $X$

# An example

- Suppose there are 100 votes, of which 12 are informal
  - $88/2+1 = 45$  votes required to win
- After the initial distribution
  - A 11, B 14, C 29, D 25, E 9
- No winner yet, so *E* is eliminated in Round 1
  - e.g. vote *EABDC* would be transferred to *A*
  - A 17, B 16, C 29, D 26
- No winner yet, so *B* is eliminated in Round 2
  - e.g. *BDAEC* goes to *D*, and *EBADC* goes to *A*
  - A 22, C 31, D 35
- No winner yet, so *A* is eliminated in Round 3
  - C 47, D 41
- C has enough votes to be declared the winner

# What types of entities do we have?

- Votes
  - Imagine a ballot paper filled in by a voter
  - Each vote is an expression of a voter's preferences
- Candidates
  - Imagine a person sitting at a table with votes that were cast for them
  - Each candidate collects votes until either they win or they're out
- Electorates
  - Imagine a big room with a group of candidates and their votes
  - Each electorate represents one constituency in the country

# The Vote class

- What is the state of a Vote?
  - The list of preferences for surviving candidates
- What inputs does the constructor need?
  - The voter's original list of preferences
- What accessor methods does Vote provide?
  - Who the Vote is currently for:

```
public char getFirstPreference(String losers)
```

- What other methods does Vote provide?
  - A test for whether the Vote is formal:

```
public boolean isFormal(String candidates)
```

# The Candidate class

- What is the state of a Candidate?
  - Their name, and their current pile of Votes
- What inputs does the constructor need?
  - Their name
- What accessor methods does Candidate provide?
  - getName(), getVotes(), getCount()
- What other methods does Candidate provide?
  - A way of adding votes to the candidate's pile:

```
public void addVotes(ArrayList<Vote> vs, String losers)
```

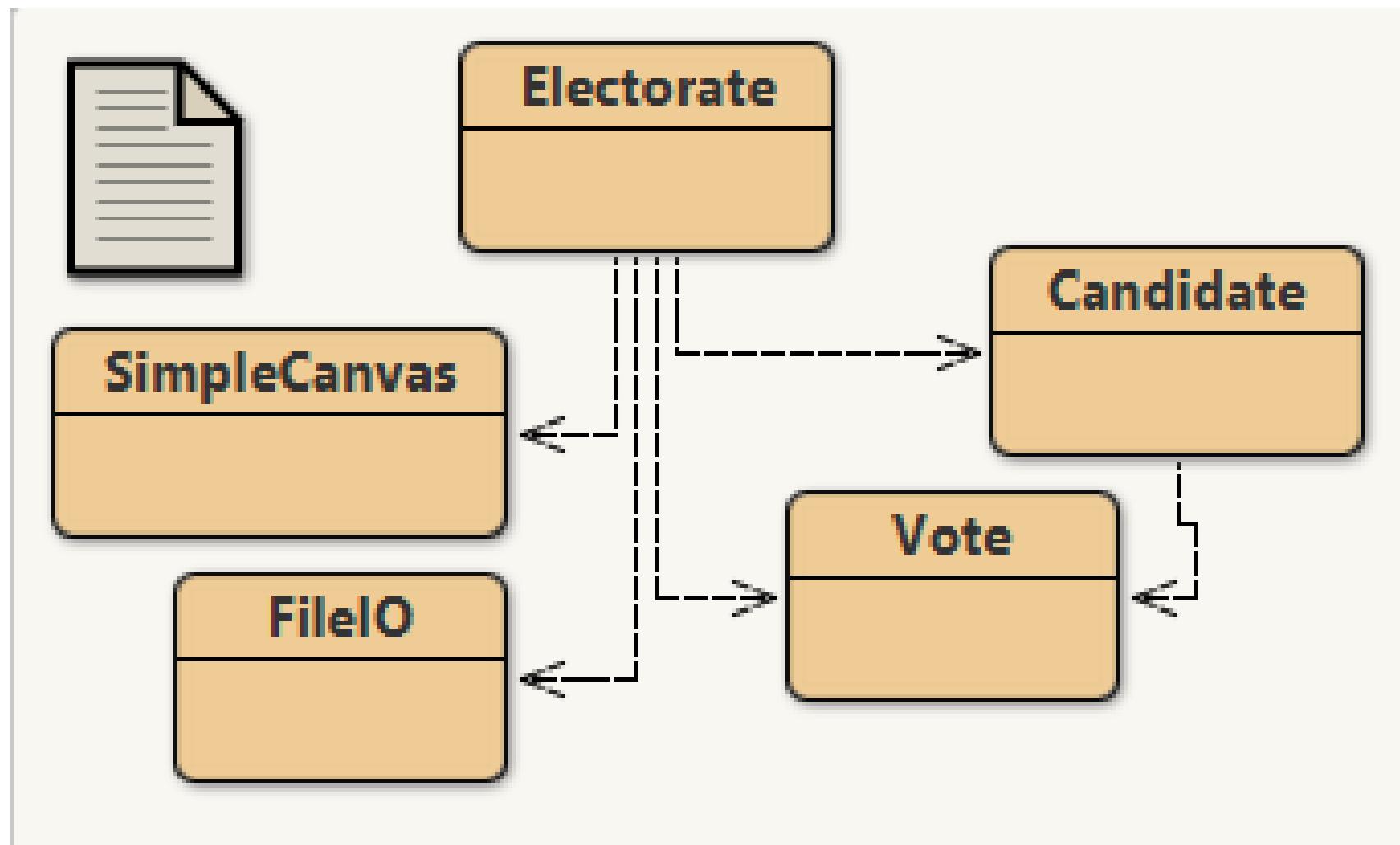
- A test for whether the candidate has won:

```
public boolean isWinner(int noOfVotes)
```

# The Electorate class

- What is the state of an Electorate?
  - A pile of Votes for each surviving Candidate
- What inputs does the constructor need?
  - Where to get the voting papers and the candidates' names from
- What accessor methods does Electorate provide?
  - None: “top-level” class, although there could be a higher class for the entire election with a collection of Electorates
- What other methods does Electorate provide?
  - The constructor does the initial distribution
  - A method to perform one round of the count
    - public void nextRound()
  - Several private methods to structure the code

# Implementation – class diagram



# Implementation questions – Vote

- What type should the list of preferences be?
  - If the candidates' names are just one `char` each, a simple `String` will do
  - Otherwise a `String[]` or an `ArrayList<String>`, depending on...
- Should we delete the names of eliminated candidates as we go along, or should we just ignore them?
  - We could ignore them when looking for the next preference
  - We could delete them as needed
  - We could delete all mentions of `c` at the time `c` is eliminated

# Implementation questions – Candidate

- What type should the pile of Votes be?
  - For most candidates, the pile will grow during the count
  - So an `ArrayList<Vote>` is best

# Implementation questions – Electorate

- What type should the pile of Candidates be?
  - Again, an ArrayList<Candidate> is best
- Should we delete eliminated Candidates, or just ignore them?
- We could imagine multiple constructors for testing:

```
public Electorate(String candidates, String votes)  
{ ... }
```

```
public Electorate(String votes)  
{this("candidates.txt", votes); }
```

```
public Electorate()  
{this("votes.txt"); }
```

# Take a look at the solution

- Available on the LMS
  - All fairly straightforward except for the display method
- Along with four sample elections
  - [small, no contest, medium, large] [candidates, votes, results]
  - SC.txt, SV.txt, SR.txt
  - NC.txt, NV.txt, NR.txt
  - MC.txt, MV.txt, MR.txt
  - LC.txt, LV.txt, LR.txt
- Just call one of the constructors for Electorate
  - Then call nextRound repeatedly

# Testing and debugging this solution

- *Testing* means checking whether a program contains any bugs
  - We can lessen the likelihood of bugs with techniques like encapsulation
  - We can make it easier to detect bugs with techniques like modularization
  - Designing intelligent test data is crucial
  - Rerun **ALL** tests after **EVERY** change
- *Debugging* means locating and fixing these bugs
  - Many debugging techniques are available
  - Remember that the cause of a bug and the manifestation of that bug can be quite separate

# An aside – bugs vs. their manifestation

- A *bug* in a program is the code where it behaves incorrectly
- The *manifestation* of that bug is where the problem becomes apparent
- Some bugs cause a program to crash immediately
  - e.g. indexing a data structure illegally
- Some bugs cause a program to crash ‘later’
  - e.g. setting up a corrupt object
- Some bugs don’t cause a crash at all
  - e.g. inputting a formula wrongly
  - Most likely this just causes the program to deliver incorrect results

# Selecting test data

- A good set of test data will include
  - Typical cases
  - Boundary cases
  - Other specifically-chosen cases
  - Error cases, where appropriate
  - Cases that give a range of results
- Where possible and appropriate, test all possible cases
  - Or close to that
- Don't be afraid of generating a **LARGE** set of test data!

# Selecting test data

- Include data that exhibits only one condition at a time
- Include data that exhibits multiple conditions
- All code must be executed at least once
  - If code hasn't been run, how can you have confidence in it?
- Don't be afraid of generating a **LARGE** set of test data!
- Your attitude must be that you **want** to break the code
  - For this reason, companies often use different people to test code from the people who wrote it

# Test data for `isFormal`

```
public boolean isFormal(String candidates)
```

- Cases that return true
  - For  $n$  candidates, there are  $n!$  possible permutations
  - If this is too many (e.g.  $n = 20$ ), sample the space systematically and/or randomly
- Cases that return false
  - Missing names, e.g. DBAC (five possible missing names), CDE (ten possible pairs), AD (ten), E (five)
  - Duplicated names, e.g. DBEACB, ACEEBDA, DADEDDBC, in all possible positions
  - Extra marks, e.g. DBACXE, YACDZEB, again in all possible positions
  - Combinations of these

# Test data for `isFormal`

- Boundary cases
  - One candidate (there still has to be an election ☺)
  - Maximum number of candidates?
  - Empty vote
  - Maximum-length vote?
- Error cases
  - Test for empty Strings, or for null
  - In both the vote and the argument

# Test data for getFirstPreference

```
public char getFirstPreference(String losers)
```

- Typical cases
  - Randomly-generated?
- Specific cases
  - No losers
  - Only two survivors
  - Only one survivor?
- Boundary cases
  - One candidate
- Error cases
  - null objects, empty Strings
  - Allow for isFormal having failed?

# Test data for entire elections

- Typical cases
  - Probably generate these randomly
- Specific cases
  - Someone wins in the first round
  - Someone leads all the way through the count
  - Someone comes from behind
  - Draws? Need to look up the AEC's rules...
- Boundary cases
  - One candidate, one vote, both
  - Candidates getting zero votes
  - One candidate getting all the votes
- Error cases
  - null objects and incorrect file names

# Debugging techniques – assertions

- Assertions can be used to get the code to check itself
- If at any point in the code we know what value a variable should have, we can insert an assertion to get that checked
- Or even if we don't know a precise value, we might know some property that the value should have
  - We might know that a number should be positive, or that an array should be sorted, or that an object variable shouldn't be `null`, or that objects should/shouldn't be aliased, or ...
- This is basically a language-specific version of inserting print statements everywhere
- Assertions allow us to identify where the code goes wrong, rather than where the program actually crashes

# Debugging techniques – breakpoints

- Setting a breakpoint in the debugger causes the program to pause at that statement, giving you an opportunity to check the values of relevant variables
- If something seems wrong, you might then
  - Proceed one step at a time, or
  - Go inside another method, or
  - Go onto the next breakpoint, or
  - Set an earlier breakpoint and restart the run, or
  - Just let the execution complete, or ...
- The BlueJ debugger provides flexible support for this process

# Debugging techniques – manual code walkthroughs

- A manual code walkthrough is basically tracing
- Many errors manifest as problems with the state of one or more objects
  - So track the states of important objects in the system
- Sometimes doing this by hand will make the reason for errors more apparent than using e.g. breakpoints
  - A breakpoint tells you merely that a value is wrong; a walkthrough may help you to identify why it is wrong

# Debugging techniques – verbal code walkthroughs

- A verbal code walkthrough is where you describe how the code works verbally, either to someone else or even to yourself
- Sometimes either the listener or the speaker will have some realization that is hard to come by in other ways
- This is really a type of abstraction; usually the code is described at different levels of detail to expose different issues
- Often used in team situations by software companies

# THE END

- This was the last lecture – basically a case study
- We will discuss exam next week

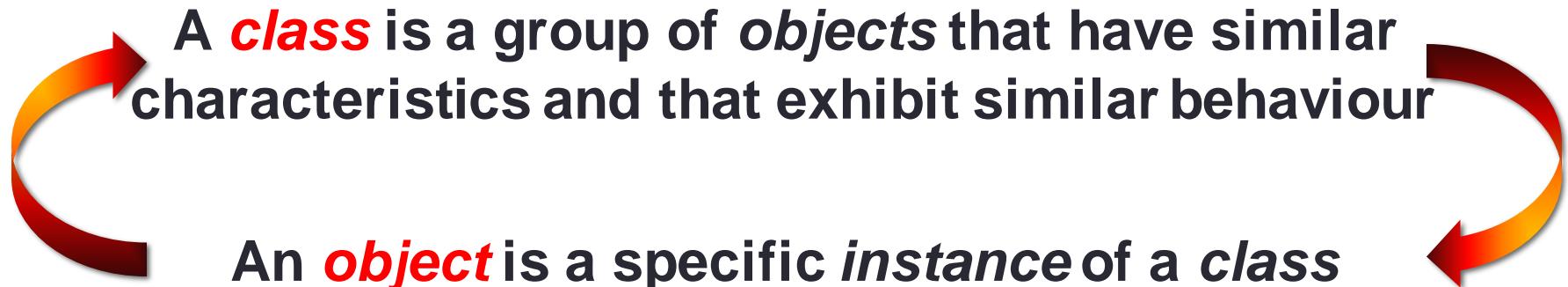
# CITS1001

## REVISION

---

# Objects and Classes

# Objects and classes



A **class** is a group of *objects* that have similar characteristics and that exhibit similar behaviour

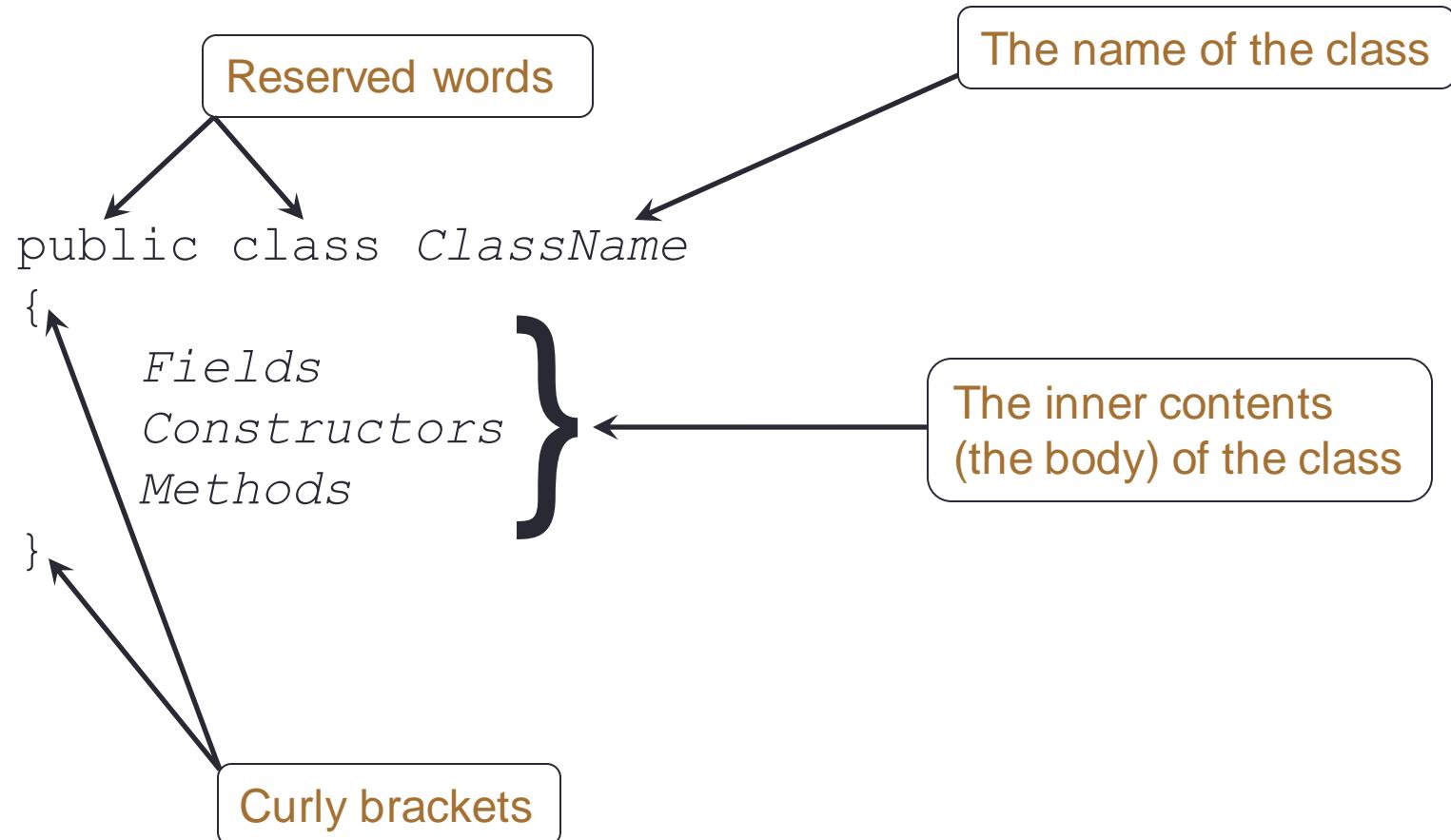
An **object** is a specific *instance* of a **class**

- Objects represent actual “things” from the real world, or from some problem domain
  - e.g. the red car in the car park
  - e.g. the lecturer talking to the group now
  - e.g. you!
- Classes represent all objects of a certain kind
  - e.g. Car, Lecturer, Student

# The four components of a class

- A class definition has four components
  - Its **name** – what is the class called?
  - Its **fields** – what information do we hold for each object, and how is it represented?
  - Its **constructors** – how are objects created?
  - Its **methods** – what can objects do, and how do they do it?
- It is (usually) best to consider the four components in this order, whether you are writing your own class, or reading someone else's
  - But during development, there may be some iteration between defining the various components

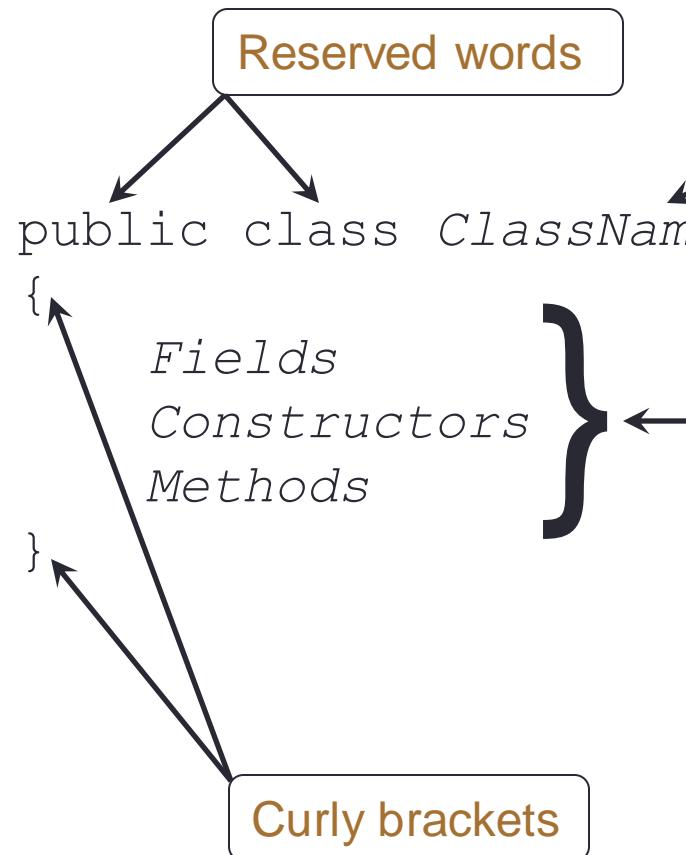
# Basic class syntax



## Class TicketMachine – naive-ticket-machine

1/2

# Basic class syntax



```

1  /**
2  * TicketMachine models a naive ticket machine that issues
3  * flat-fare tickets.
4  * The price of a ticket is specified via the constructor.
5  * It is a naive machine in the sense that it trusts its users
6  * to insert enough money before trying to print a ticket.
7  * It also assumes that users enter sensible amounts.
8  *
9  * @author David J. Barnes and Michael Kölling
10 * @version 2016.02.29
11 */
12 public class TicketMachine
13 {
14     // The price of a ticket from this machine.
15     private int price;
16     // The amount of money entered by a customer so far.
17     private int balance;
18     // The total amount of money collected by this machine.
19     private int total;
20
21     /**
22      * Create a machine that issues tickets of the given price.
23      * Note that the price must be greater than zero, and there
24      * are no checks to ensure this.
25      */
26     public TicketMachine(int cost)
27     {
28         price = cost;
29         balance = 0;
30         total = 0;
31     }
32
33     /**
34      * Return the price of a ticket.
35      */
36     public int getPrice()
37     {
38         return price;
39     }
40
41     /**
42      * Return the amount of money already inserted for the
43      * next ticket.
44      */
45     public int getBalance()
46     {
47         return balance;
48     }
49

```

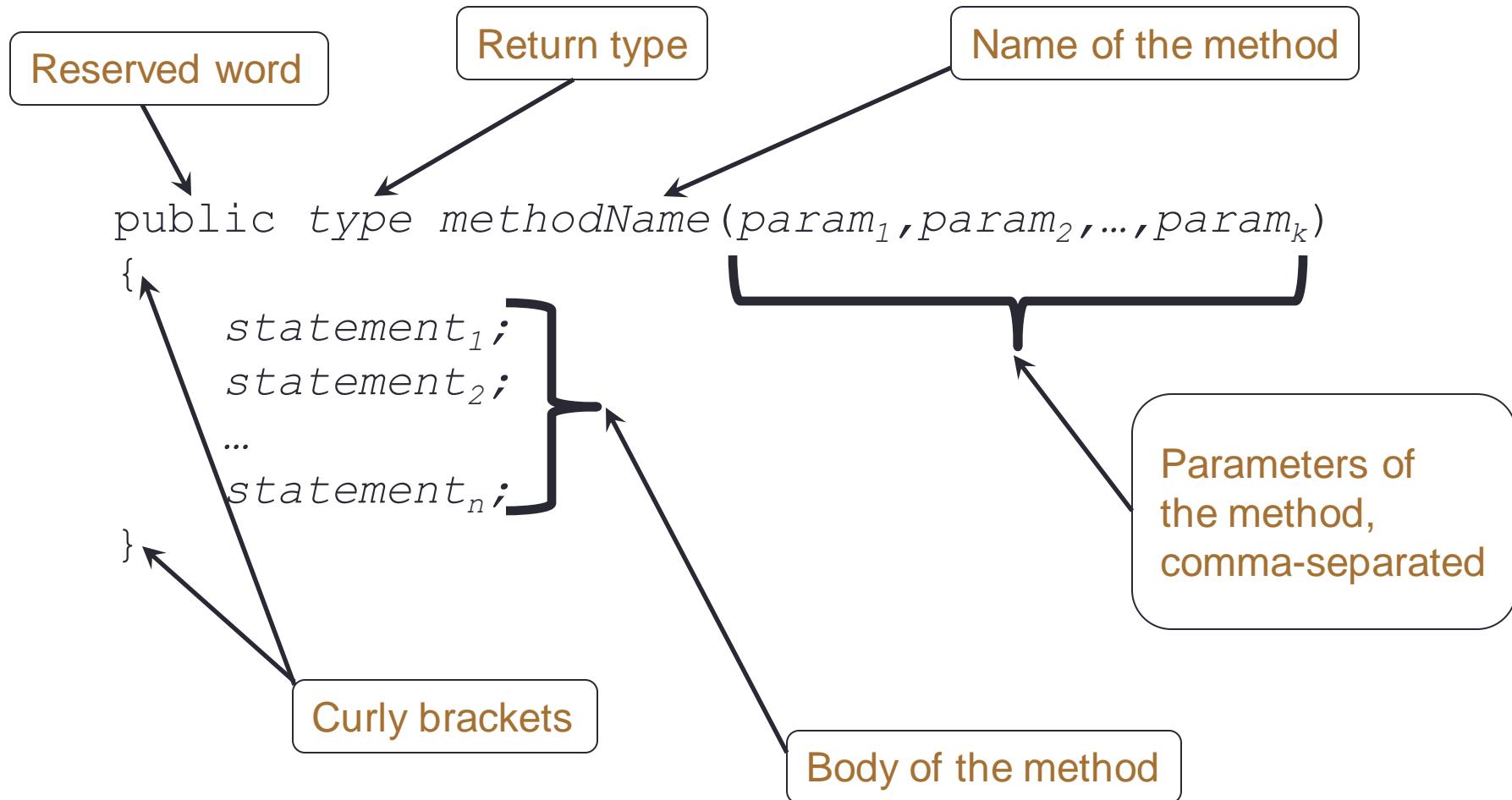
# Syntax

- Reserved words and curly brackets are our first encounter with *Java syntax*
- Source code must be structured in a certain way, as determined by the rules of the language
- Reserved words are words with a special meaning in Java
  - e.g. public, class, private, int, String
  - There are many, many others that we will meet as we go along
  - Also known as *keywords*
- Brackets (of all types) are everywhere in many languages
  - Basically used to group things together
  - Here, the curly brackets delimit the body of the given class

# Methods

- Methods implement the behaviour of objects
- Methods can implement any form of behaviour, as required by the class being implemented
  - The principal value of computers lies in their flexibility
  - A method comprises a *header* and a *body*

# General method syntax



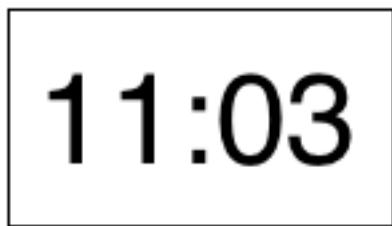
# Example methods

- We will introduce methods first via two relatively simple specific forms that are very common in Java programs
- *Mutator methods* alter the state of an object
- *Accessor methods* provide information about an object
- Remember that other sorts of methods can accomplish a very wide variety of tasks

# Abstraction and modularization

- *Abstraction* means ignoring the details or the parts of a problem, to focus attention on its higher levels
  - cf. User-view vs. writer-view of a class
  - The writer must worry about details; the user cares only how to use the class and what it does
- *Modularization* or *decomposition* means dividing a whole into well-defined parts, which
  - Can be built and examined separately
  - Interact in well-defined ways
  - Can be treated as “units” at higher levels of the design
  - cf. car design

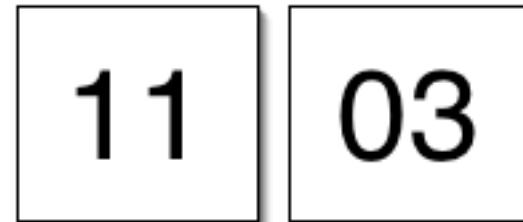
# Modularizing the clock display



One 4-digit display?

- We would need to solve all of the clock issues in one context

Or two 2-digit displays?



- How would the behaviour of these two displays compare?

# Variables

- Variables are the basic mechanism by which data is organised and stored in all programming languages
  - Used for long-term storage, short-term storage, and communication

*type* **variableName** = value;

- Every variable in a Java program has a declaration, which determines
  - Its name, by which it is accessed and updated
  - Its type, which determines what values it can store and what operations it can participate in
  - Its scope, which determines where in the program it can be used; this is implicit from what kind of variable it is.

# Repetition

# Iteration fundamentals

- We often want to repeat some actions over and over
  - e.g. “do this action for each student in CITS1001”
  - e.g. “do this action a hundred times”
  - e.g. “do this action until the room is empty”
- Loops provide us with a way to repeat actions, and to control how many times we repeat those actions
- The first paradigm above is done by the *for-each* loop
- The second paradigm is done by the *for* loop
- The third paradigm is done by the *while* loop
  - Also the *do-while* loop

# Templates of the various loops

```
for (Type element : collection) {  
    statements to be repeated  
}
```

---

```
for (initialization; condition; post-body action) {  
    statements to be repeated  
}
```

---

```
while (condition) {  
    statements to be repeated  
}
```

---

```
do {  
    statements to be repeated  
} while (condition);
```

# Libraries

# Libraries

- The libraries available in Java are accessible from the Java API

<https://docs.oracle.com/en/java/javase/11/docs/api/>

- There are many, many classes
  - Organised into *packages* of similar material
- Use built-in types and methods whenever they are suitable
  - If someone has already written the code that you need, using their class will usually be more efficient than writing your own
- Example: String, ArrayList, Random, *etc.*

# Importing classes

- Classes or packages can be imported explicitly, e.g.

```
import java.util.Random;  
import java.util.*;
```

- This enables you to use the short name in your code, e.g.

```
Random r = new Random();
```

- Some library classes are imported automatically
  - Including many that you may use frequently, e.g. Math, String, Integer, Character, Boolean, etc.

# Collections

# Fixed-size collections

- An `ArrayList` is used when the size of a collection is not known in advance, or might vary
- But in some situations, the collection size can be pre-determined from the data
- For this situation, a special fixed-size collection type is available: the `array`
  
- Arrays can store object references or primitive values
- Arrays use a special more-concise syntax
- Arrays are very common in a wide range of programming languages

# Uses of arrays

- Arrays are used when we have large numbers of same-typed values or objects that we want to operate on as a collection
  - A collection of temperatures that we want to average
  - A collection of student marks that we want to analyse
  - A collection of names that we want to sort
- e.g. Bureau of Meteorology monthly data
  - We know there are twelve months in a year

## ALBANY

Max	25.1	25.1	24.1	21.5	18.7	16.6	15.7	15.9	17.4	18.8	20.8	23.4
Min	13.5	14.3	13.3	11.6	9.8	8.1	7.4	7.4	7.9	9.0	10.6	12.3
Rain	28	25	29	66	102	104	126	104	81	80	46	24

## PERTH AIRPORT

Max	31.4	31.7	29.5	25.2	21.4	18.7	17.6	18.3	20.0	22.3	25.4	28.5
Min	16.7	17.4	15.7	12.7	10.2	9.0	8.0	7.9	8.8	10.1	12.4	14.6
Rain	8	14	15	46	108	175	164	117	68	48	25	12

# Declaring arrays

- An array variable is declared using specific syntax
  - The [ ] denotes an array variable

```
int[] a;
```

- Declares `a` to be a variable representing an array of `ints`

```
double[] temps;
```

- Declares `temps` to be a variable representing an array of `doubles`

```
String[] names;
```

- Declares `names` to be a variable representing an array of `Strings`

```
Student[] marks;
```

- Declares `marks` to be a variable representing an array of `Students`

# Arrays

```
Student[] unit;
```

*Declare*

```
unit = new Student[numStudents];
```

*Create*

```
unit[0] = new Student("042371X", 64);
```

```
unit[1] = new Student("0499731", 72);
```

```
unit[2] = new Student("0400127", 55);
```

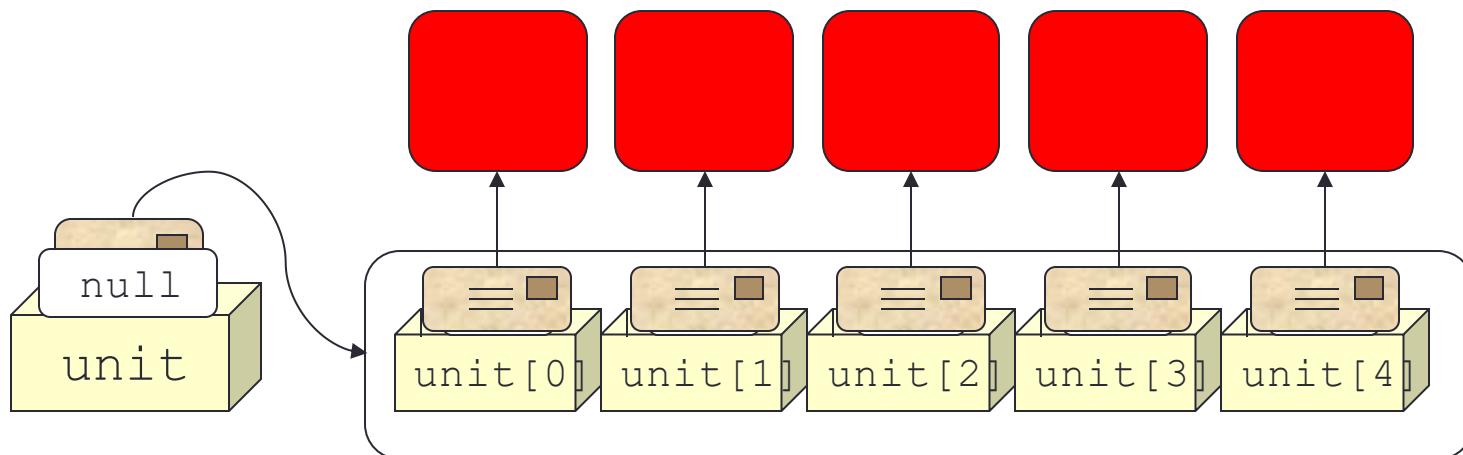
*Populate*

...

```
unit[numStudents-1] = new Student("0401332", 85);
```

# The three steps

- Declare – `int[] a;`
- Create
  - Using `new`
    - `a = new int[5];`
- Populate (repeatedly)
  - *Each object needs `new` to be executed*



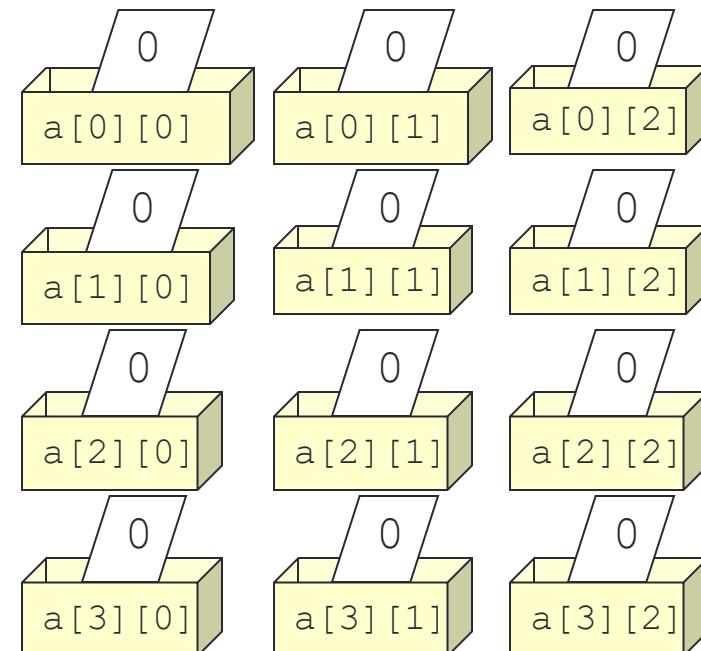
# 2D arrays

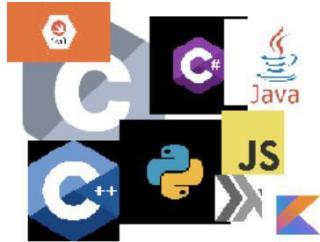
- We sometimes need an array with more than one dimension

```
int[][] a = new int[4][3];
```

This creates an array with four “rows” and three “columns”

The “row” index ranges from 0–3 and the “column” index from 0–2





Arrays start at 0



Arrays start at 1



Perl

Arrays can start  
wherever ^\\_(ツ)\_/^-



Arrays start at 4,  
stop at 6, restart  
at 1, stop again  
at 3, restart at 7  
then continue on



# Enumerated Types

# Declaring variables

```
int profit; // holds the annual profit of a  
company  
int price; // holds the price of a car  
int mark; // holds someone's mark in a unit  
int date; // holds the day of the month
```

- Also the likely *ranges* of these variables are very different
  - profit can be any value, positive or negative
  - price can only be positive
  - mark can only be in the range 0..100
  - date can only be in the range 1..31

# Discrete data with a restricted range

- How can we represent this kind of data in a Java program?
  - There are two common ways
- Using int
  - Choose a number to represent each value,  
e.g. Monday = 1, Tuesday = 2, Wednesday = 3,  
Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7
- Using String
  - Use the names (or parts of them),  
e.g. "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"
- Such mappings are often called *encodings*

# Discrete data using int

- This method returns the number of hours worked by a casual tutor on a given day of the week

```
public int noOfHours(int day)
{
    if (day == 1) return 2;
    if (day == 2) return 5;
    if (day == 3) return 3;
    if (day == 4) return 6;
    if (day == 5) return 6;
    return 0;
}
```

# Discrete data using String

- This method returns the number of hours worked by a casual tutor on a given day of the week

```
public int noOfHours(String day)
{
    if (day == "Mon") return 2;
    if (day == "Tue") return 5;
    if (day == "Wed") return 3;
    if (day == "Thu") return 6;
    if (day == "Fri") return 6;
    return 0;
}
```

# Problems

- This approach has at least four serious problems
- What if a programmer or user enters an ‘illegal’ int/String?
  - e.g. 8, or “Thi”?
  - All methods must be written to allow for this possibility
- Will programmers and users remember the encoding?
  - e.g. was Monday 0, or 1?
  - e.g. was Thursday “Thursday”, or “Thu”, or “THU”?
- It is easy for programmers to inadvertently swap variables or values that are being used for different encodings
  - e.g. if you’re using `int d` for days and `int m` for months
- There’s nothing to stop programmers (accidentally or deliberately) using type-correct operations on days
  - e.g. adding `int` days together
  - e.g. taking a substring of a `String` day

# Enumerated Types

- Java provides a special (and very simple!) type of class called the `enum`
- Also known as *enum* types
- Special **data type** that enables for a variable to be a set of predefined constants.
- An example:

```
public enum Day  
{MON, TUE, WED, THU, FRI, SAT, SUN}
```

- Day is now a new type in the program
- The values are accessed as e.g. `Day.MON`, `Day.TUE`
  - They are written using capitals because they are constants

# enum comes with built-in facilities

- Days can be compared for equality and inequality

```
Day d1, d2;  
if (d1 == d2) {} {}
```

- They can be added to Strings

```
Day d = Day.TUE;  
String s = "I love " + d; // sets s to "I love TUE"
```

- The range can be accessed easily...

- Day.values() denotes the Day[] array  
{Day.MON, Day.TUE, ..., Day.SUN}

- ...so they can be processed using a for-each loop

```
for (Day d : Day.values()) {}
```

- They can be ordered using the method ordinal()

```
int x = d.ordinal(); // sets x to 1
```

# Encapsulation

# Encapsulation

- One of the most important features of object-oriented programming is that it facilitates *encapsulation*
- This means that a class describes both the data it uses, and the methods used to manipulate that data
- The external user sees *only* the public methods of the class, and interacts with objects belonging to that class purely by calling those methods
- This has several benefits
  - Users of the class can call the public methods without needing to understand their implementation or the representation of the data
  - Programmers can alter or improve the implementation of the class without affecting any client code
  - Use and implementation are *divorced*
- Abstraction again!

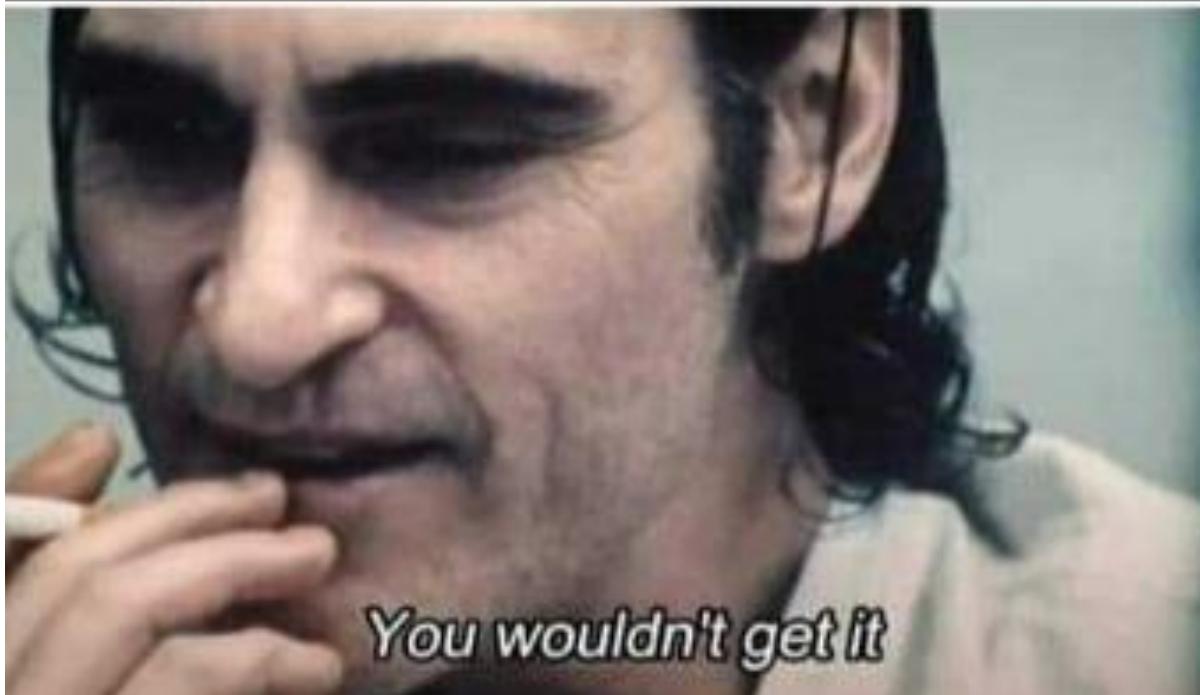
# Access restrictions

- Encapsulation is enforced by the correct use of access modifiers on instance variables and methods
  - `public`, `private`, `<default>`, and `protected`
- If you omit the access modifier, you get the default, sometimes known as “package”
- The latter two modifiers are relevant only for multi-package programs that use inheritance, so at the moment we need consider only `public` and `private`

# public and private

- If an instance variable is public
  - Any object can access it directly
  - Any object can alter it directly
- If an instance variable is private
  - Only objects that belong to *the same class* can access and alter it
- If a method is public
  - Any object can call that method
- If a method is private
  - Only objects that belong to *the same class* can call it
- Note that privacy is a per-class attribute, not per-object

```
1 public class Meme
2 {
3     private Joke joke;
4
5     public void setJoke(Joke newJoke)
6     {
7         this.joke = newJoke;
8     }
9 }
```



*You wouldn't get it*

# Exceptions

# Dealing with errors

- Even if your classes are well-protected, errors will occur
  - *All* non-trivial software contains errors!
- We consider three types of error
  - Client code attempts to use your methods incorrectly, by passing incorrect parameter values
  - Your code cannot perform the services it is meant to, due to circumstances outside your control (e.g. an Internet site is unavailable)
  - Your own code behaves incorrectly and/or your objects become corrupted
- To handle these problems, Java provides
  - *Unchecked exceptions*
  - *Checked exceptions*
  - *Assertions*

# Operation of try/catch

- Logically, try/catch operates a lot like if/else
- If everything goes smoothly
  - The code in the try block is executed, and
  - The code in the catch block is skipped
- If *any* of the statements in the try block causes an exception to be thrown
  - Execution immediately jumps to the catch block, which tries to recover from the problem
- What can the catch block do?
  - For human users: report the error and ask the user to change their request, or retype their password, or ...
  - In all cases: provide some feedback as to the likely cause of the error and how it may be overcome, even if ultimately it just causes execution to cease

# Multiple catch blocks

- Sometimes the `try` block might call several methods that can throw different checked exceptions
  - Or even one method that can throw several different checked exceptions...
- In this case multiple catch blocks are required
  - One to deal with each possible checked exception, e.g.

```
try {  
    // code for when everything is OK, but which  
    // might throw multiple checked exceptions  
}  
catch (java.io.FileNotFoundException e) {  
    // code for when things go wrong  
}  
catch (java.io.URISyntaxException e) {  
    // code for when things go wrong  
}
```