CITS 1200 Java Programming

Computing Fundamentals
Scope of this lecture

- Hardware
- Software
- Bits and Bytes
- Central Processing Unit
- Programming Languages
- Java Virtual Machine
Computer Hardware

• Physical machine, comprising
  – *central processing unit* does the actual calculations
    • eg Pentium, AMD Athlon, G5
  – *random access memory* (RAM) for quickly accessible workspace
    • typically 1Gb to 4Gb
  – *disk memory* which is slower but far larger for data storage
    • typically 120Gb to 1Tb
  – *keyboard* and *mouse* for input
  – *monitor* and *printer* for output
  – large variety of accessories
    • BluRay/DVD/CD drive, USB/FireWire, cameras, speakers etc
Computer Software

• Instructions, in the form of programs, that make the computer hardware do useful things.
  – An operating system performs the low-level operations managing the computer’s physical resources
    • e.g. Linux, Windows XP/Vista, Mac OS X
  – A window manager controls the look and feel of the graphical user interface (GUI) connecting the user and the operating system
    • e.g. Gnome, KDE (both for Linux)
  – Application software is the generic term for programs that perform the tasks actually required by the user
    • e.g. Word, Firefox, BlueJ, Forte
Modern computers are all digital computers which store and manipulate data in the form of binary digits or bits

- A single bit stores one of only two values 0 or 1
- To store more complicated data we use more bits
  - e.g. two bits can store four different values 00 01 10 11
  - Using $n$ bits the number of values that can be stored is $2^n$
- Bits are grouped together into bytes, each containing 8 bits

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

- Computer memory is just a long series of bytes

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]
Central Processing Unit (CPU)

- The central processing unit is the “brain” of the computer
- Can perform a limited number of very simple instructions
  - moving a byte from one location to another
  - adding two bytes
  - comparing two bytes
  - jumping to another location in memory
- The CPU fetches each memory location in turn, and executes the simple instruction contained at that location
  - the clock speed is the speed at which this “fetch-execute” cycle happens; typical numbers are 1.25GHz – 4 GHz
Programming Languages

- Programming in the machine language needed by the computer is lengthy, tedious and error-prone, and each type of computer has a different machine language.
- A high-level programming language allows users to specify more naturally the actions for the computer to take.
- Languages like C and C++ are compiled into machine code.

```
C code
printf("hi");
```

compilation diagram:

```
C code
compiler
Machine code
```
Java Virtual Machine

- Java compiles to bytecode
  - intermediate format
  - not machine code for any real machine
- Bytecode is machine code for the *Java Virtual Machine*
  - to run on a real machine need a Java interpreter
  - bytecode can be transported from one machine to another
Java and the Web

• One of the reasons for Java’s popularity is its connections to the Web
• A Java applet is a Java program that is run by a Web browser such as Firefox, Safari or Internet Explorer
• The writer of a Web page writes the Java program, compiles it into bytecode and places it on a Web server
• The remote Web browser then downloads the bytecode and interprets it, thus running the program using the resources of the client machine
• We will not be using applets in this course, although keen users are welcome to read about them in Barnes
Programming Paradigms

- Different programming languages encourage the use of particular programming *styles* or *paradigms*
- The Java programming language is an object-oriented language so it encourages programs to be structured in an object oriented fashion
- If you have previously used other languages, you may find that you need to adjust your programming style
There are two important aspects to programming

• Firstly you have to specify what you want to achieve so precisely and so completely that even a dumb computer can do the job for you. This is hard.

• Secondly you want to ensure that your programs are efficient in terms of speed and memory usage.
The Hair Washing Algorithm

Shampoo bottles have a set of instructions, an algorithm, for washing your hair. These instructions illustrate some of the common problems in programming.

1. Shake well
2. Wet hair thoroughly
3. Apply Shampoo
4. Lather
5. Rinse
6. Repeat
1) Shake well
   What do we shake - yourself?
   How long before it is considered well shaken?

2) Wet hair thoroughly
   Wet your hair with what - coffee?
   Where should you do the wetting? - in bed, in the bath or shower?
   What about the rest of your body, should you take your clothes off first, or wear a raincoat to keep the rest of you dry?

3) Apply Shampoo
   Apply it to what...
   How much do you apply, the whole container?
4) Lather
   What? - presumably the shampoo in your hair.
   For how long?

5) Rinse
   What? - presumably rinse out the shampoo in your hair.
   What with - more coffee?

6) Repeat
   Repeat what? Not starting at step 1 - the shampoo is already shaken.
   Not at step 2 - your hair is already wet.
   We actually repeat starting at step 3.
4) Lather
   What? - presumably the shampoo in your hair.
   For how long?

5) Rinse
   What? - presumably rinse out the shampoo in your hair.
   What with - more coffee?

6) Repeat
   Repeat what? Not starting at step 1 - the shampoo is already shaken.
   Not at step 2 - your hair is already wet.
   We actually repeat starting at step 3.
   How do we stop repeating?
Efficiency

Think of what an internet search engine has to do...
Searching for a name in a list - assume there are N names are kept in a long array.

If the names are randomly stored in the array you will have to test every single entry to ensure that you find all occurrences of the name.

You will need to perform N tests.
If the names are stored alphabetically in the array a sequential search for the name can be stopped as soon as you encounter a name that comes after the name you want.

```
A A E F G I L L M N N O O S S T
```

On average you will stop half way down the list, so on average you will need to perform N/2 tests - twice as fast.
If the names are stored alphabetically you can perform what is called a binary search.
Check the name that is half way down the list. If it comes after the name you want you can ignore the right half of the list and just consider the left half.
We then repeat the process with the remaining left half. If the name in the middle comes before the one you want you can discard the left half of the remaining list, and so on.

A  A  E  F  G  I  L  L  M  N  N  O  O  S  S  T
Find the letter I

```
A  A  E  F  G  I  L  L  M  N  N  O  O  S  S  T
```

 ^=  ^=  ^=  ^=

2  4  3  1

The numbers indicate the order in which entries are searched in order to find ‘I’. In each step of a binary search you are dividing the list of names in half, thus in the worst case you will need $\log_2 N$ (log to base 2) tests to find the name.
• Having a process that requires time proportional to log(N) rather than time proportional to N can result in huge speed gains.

• The population of Australia is about 20,000,000 - but this will only require about 25 tests to find a name from the Australian population using a binary search.
CITS 1200 Java Programming

Computer Programming
Scope of this lecture

- What is a program?
- What is a computer program?
- The software crisis
- Programming in the small
- Programming in the large
- Object-oriented programming
- Forces favouring OOP
- Some history
What is a program?

• A sequence of elementary instructions that when followed achieves a specified goal
  – a procedure, a recipe, an algorithm

  2. Stir fry 1 clove garlic, 1 tbsp black beans, 100g preferred roast meat.
  3. Add 2 small bok-choy and toss to coat.
  4. Scatter a pinch of sugar and add 2 tbsp water.
  5. Reduce heat, cover and cook for two minutes.

Stephanie Alexander, *the cook’s companion*
Elementary instructions

• Is “stir-fry” an elementary instruction?

subroutine stirfry

1. Heat the wok.
2. Add enough oil to hot wok to generously coat food.
3. Splash hot oil up sides of wok to coat entire surface.
4. Add vegetables in order they will take to cook.
5. Toss contents.
6. When all pieces are coated, hissing and sizzling add a little water, and cover.
7. Remove after three minutes.

Stephanie Alexander, *the cook’s companion*
What is a computer program?

• Simple version
  – a sequence of elementary instructions for a computer to follow

1. Ask user for a number and call it F
2. Subtract 32 from F
3. Divide F by 1.8 and call the answer C
4. Tell the user the number C

INPUT F
F = F - 32
C = F/1.8
OUTPUT C

Warning: This is not Java code
Dissection of this program

• Each line is an instruction for the computer to follow
  • INPUT F
    – means “Get a number from user and call it F”
    – F is called a variable, because it is the “name” of a value that can be changed while the program runs
  • F = F – 32
    – Looks strange to the mathematically inclined
    – The F–32 part is called an expression
      • computer looks up the value that F refers to, subtracts 32, and holds answer in a temporary location
    – The F = .. part is called an assignment
      • computer overwrites old value of F with the new calculated value
Dissection of this program (cont)

- \( C = \frac{F}{1.8} \)
  - Easy for us to interpret now
  - Computer looks up value of \( F \), divides it by 1.8, and stores it in a new location called \( C \)

- **OUTPUT** \( C \)
  - Make the value of \( C \) appear to the user
  - (Usually just on the user’s monitor)

- The specific numbers that appear in the program – the 1.8 and the 32 – are called *literals*
Programming Languages

• 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 that you can write statements that can be understood
  – how to make sequences of legal statements that do simple tasks
  – how to express what you wish 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 when learning a new human language
Same program, but in C

```
#include <stdio.h>
main() {
    double c;
    double f;

    scanf("%lf",&f);

    f = f-32;
    c = f/1.8;

    printf("%f\n",c);
}
```

C insists that we declare variables before using them

Basically the same program, but with a different syntax
Compile time and runtime

- What the programmer writes is called *source code* or *code*
- Although much more restricted and simple than human language, source code cannot be executed directly by the computer, but must be further translated
- Java source code is *compiled* into an even simpler form (byte-code) by the *Java compiler* – the Java compiler checks that the source code is *legal*, and goes to some effort to make sure that it is *sensible*
- If compilation succeeds, then the program can actually be *executed*, or *run* – this stage is called *runtime*

*A program may compile but still do the wrong thing!*
Programming in the small

- Single programmer who understands entire program
- Program performs a small pre-specified range of tasks
- Programs often take input, perform calculations, return results and stop
  - e.g. program that takes in an amount, an interest rate, and a time period and calculates the repayment amount on a mortgage

- Design considerations are not so important with very small programs, and moderately sized programs are often best designed using structured programming
Structured Programming

- Also known as *top-down design*
- Divide the main task into sequence of sub-tasks
- Some of the sub-tasks may be further subdivided
The software crisis

• In 1968, it was observed that almost all software projects are delivered over budget, behind deadline and do not meet specifications
• This was termed “the software crisis”
• A 1995 study reported
  – Five out of six corporate software development projects are considered unsuccessful
  – One third of all corporate software development projects are cancelled
• Over 70% of the cost of software is spent on software maintenance and/or software preservation
Why the crisis?

• The two major factors behind the software crisis are *complexity* and *change*
• Software, largely because of its dynamic nature, is one of the most *complex* things that humans construct
• *Change* is ever-present
  – User requirements are not fully specified
  – Users change their minds
  – Changes are forced due to constraints in budgets, resources, timescales, equipment, system performance (etc)
  – Using the system leads to *new* requirements
Software goals

• **Correctness** and **robustness**: software products should perform as specified, and be robust to unexpected conditions
  – This requires managing **complexity**

• **Extendibility** and **reusability**: software should be extendible so that it can be changed, and re-usable so that we do not always have to start from scratch
  – This requires managing **change**

• Structured Programming cannot cope with extreme complexity or rapid change
Programming in the large

• Most computer programs these days
  – run continuously
  – respond to input from users and other programs
  – deliver output to users and other programs
  – are programmed by large teams of programmers
  – are too big and complex for any one person to understand

• For example, any Web browser
  – responds to mouse clicks, and text entries
  – communicates with computers across the Internet
  – formats and displays Web pages
  – starts up “helper applications” if necessary
Complexity in the real world

• How is something moderately complex organized in the real world? Consider a restaurant ...
  – Diners want meals
  – Chefs prepare dishes
  – Waiters take orders from diners, and bring food to the tables
  – Busboys collect and wash plates
  – Barmen prepare and serve drinks
  – The maitre’d makes reservations and seats diners

Each type of person provides a narrow range of services. The restaurant operates through the co-operative interaction of all the restaurant staff
Object Oriented Programming

- OOP refers to the process of designing and implementing a co-operative community of interacting objects.
  - each object provides a small number of relatively simple services
  - objects communicate with each other to exchange information
  - complex behaviour is achieved by the co-operation of objects

Diner
Waiter
Chef

Bob
Pierre
Joe

Bok-choy with black beans, please
One number 23

© CSSE, Gordon Royle 2008
What does OOP provide?

• Complexity management
  – *encapsulation* and *information hiding* work on the principle that objects should interact *only* through pre-specified *interfaces*
  – software development can be more reliably divided between independent groups

• Reusable software
  – *class libraries* provide easy access to many standard services
  – developing *software components* that match the reliability and interchangeability of hardware components is an elusive goal

• Natural modelling
  – problem identification, program design and program implementation all follow same process
External forces favouring OOP

• Client-server paradigm
  – most modern computing is based on servers providing services to clients so this extends naturally to programs

• Event driven programming
  – actions occur in response to external events, with the program being otherwise idle

• Increasing use of simulation
  – simulation of physical, natural or human situations deals with complex interactions of large numbers of relatively simple “atoms”

• Easily adaptable to parallel computing
  – two or more CPUs can operate on the same collection of agents with almost no behavioural change
Difficulties with OOP

- OOP involves sophisticated concepts, and needs sophisticated language to express them properly
- The benefits of OOP are often not apparent for the small programs possible in a beginning programming course
- OOP is indirect compared to “traditional” programming
  - In traditional programming, source code statements are simply executed sequentially by “the computer” and so there is a direct relationship between the source code and the runtime
  - In OOP, source code statements are executed by “objects” of which arbitrarily many can be created and destroyed at runtime
Some C-like languages

- A number of important modern languages have their roots in the C programming language from the 1970s
- C is a fast and powerful language that gives the programmer complete control over almost every aspect of their code
  - It also requires programmers to manually manage many low-level aspects of their code, particularly memory management; this makes it prone to certain types of bug, and somewhat difficult to scale up
- C++ is a successor of C which adds object-oriented ideas to C, while retaining the speed and control of C
  - The object oriented features of C++ are optional, which means that many programmers write “C code in C++”
  - The combination of features makes C++ a difficult language to learn
More C-like languages

• Java has syntax that heavily borrows from C and object-oriented features like those of C++, but it enforces the object-oriented paradigm much more rigidly
  – Java has automatic memory management which eliminates a large percentage of frustratingly difficult-to-find bugs
  – Java has effective information-hiding mechanisms which make it easier to write code as separate modules
  – There is a performance cost for this, and Java code is significantly slower than the equivalent C or C++ code, although this is highly problem dependent

• C# (pronounced C-sharp) is a Microsoft language very similar to Java, but with features to integrate with .NET.
Why Java?

• Java is largely object-oriented and its structure strongly encourages programs to be written in that paradigm
  – However, it is still possible to write Java code in a procedural fashion and there are a number of very poor Java texts that do this
• Java is much simpler to learn than C++ because it automatically manages low-level aspects (particularly memory) that bog down beginners in C++
• Java is freely available for the three major platforms today (Windows, MacOS and Linux/Unix) and code written in Java is relatively portable
• Java programmers can learn both C# and C++ quickly
CITS 1200 Java Programming

Classes and Objects
Scope of this lecture

• Definition of classes and objects
• Examples of classes and objects
• Creating objects in BlueJ
• Inspecting objects in BlueJ
A class is a group of objects that have similar characteristics and exhibit similar behaviour.

An object is a specific instance of a class.
Real-life examples

• Although the definition is daunting, we are very familiar with classes in real-life

• Example
  – The set of all dogs forms the class Dog
  – Each individual dog is an object of the class Dog
  – Spot, Rover and Rex are all instances of the class Dog

• Example
  – The set of all students forms the class Student
  – Each individual student is an object of the class Student
  – John Smith and Janice Lee are instances of the class Student
Why do we use classes?

- We use classes to reduce complexity
- We can often deal with a specific object based purely on knowledge of its *class*, rather than that particular *instance*
- For example, if we encounter a dog – in other words, an instance of the class `Dog` – walking down the street, we already have a basic understanding of how to deal with it, even if we have never met that *particular* dog
Describing a class

- We describe a class by listing the common features that are shared by all the objects in that class, divided into the attributes that each object has, and the actions that each object can perform.
- Example:
  - Student number is an attribute of the class Student.
  - Every student has a student number, although each individual student has a different student number.
- Example
  - Barking is an action that all objects of the class Dog have in common.
What is a Waiter?

• A **Waiter** has the following *attributes and actions*
  – Attributes of a Waiter
    • Name
  – Actions of a Waiter
    • Bring menus
    • Take orders
    • Bring meals

• This collection of attributes and actions *defines* the class of **Waiters**
  – We can deal with any individual waiter just based on our knowledge of the class **Waiter**
Symbolic class description

- Classes are often described by giving a symbolic representation of the following form:

A rectangle divided into three parts containing the name of the class, the list of attributes and the list of actions.
How is this related to Java?

• The idea behind object oriented programming is that a program’s source code should describe the classes involved in the problem domain.

• At runtime, objects of those classes are created and they behave according to the source code describing the class to which they belong.

• Complex programs can arise by the interaction of many objects from different classes, even though each individual class description may be quite simple.
A simple example

- Banks use financial software to maintain their records
- The OOP approach is to consider the classes of objects in the problem domain, and then to write source code to represent those classes
- This is quite different from “traditional programming” where the first question would be “What should the program do?”
- For the banking example, an obvious class that will be needed is one to represent bank accounts
The class **BankAccount**

- A bank account has the following attributes
  - An *account number* and an *account name*
  - A *balance*

- A bank account has the following *actions*
  - Money can be *deposited* into the bank account
  - Money can be *withdrawn* from the bank account

<table>
<thead>
<tr>
<th>BankAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance</td>
</tr>
<tr>
<td>account Name</td>
</tr>
<tr>
<td>account Number</td>
</tr>
<tr>
<td>deposit</td>
</tr>
<tr>
<td>withdraw</td>
</tr>
</tbody>
</table>
Another analogy is that OOP is a bit like a role-playing game such as dungeons and dragons, which has classes of characters, such as gnomes, trolls, wizards etc.

The objects in a particular class, say Wizard, can do certain “wizardy” things like casting spells or turning invisible - these are the actions.

Although wizards are all basically similar, individual wizards may differ because as the game progresses they may gain strength or knowledge.
Another Example

- A building designer might run a computer simulation of the activity within a large building to test the effects of varying the parameters in the design.
- The OOP approach would be to design classes to represent the various components of a large building – elevators, stairs, workers, offices, doors etc.
- At runtime, a number of objects from these classes would be created, and the simulation would consider how they would interact – for example, we might want to know how many elevators are needed to ensure that the queues at morning rush hour are not excessive.
The class Elevator

• There are many choices in designing a class
• We want to include relevant items – the capacity of elevator for example, while ignoring irrelevant ones such as its interior colour
• Some possible attributes of an Elevator object include
  – currentFloor
  – currentPassengers
  – capacity
• The values of some of the attributes may change during runtime, while some others may remain fixed
  – currentFloor will be changing, while capacity will not
Actions of an Elevator

• What are the relevant actions of an Elevator?
• In a simple simulation, we may just include the most fundamental ones
  – goUp
  – goDown
  – addPassengers
  – dropPassengers
• The source code for this class must define carefully the behaviour of an Elevator so that it is an accurate model for a real elevator
  – It cannot addPassengers if it is already full
  – It cannot goUp if it is on the top-floor
Objects and Classes in Java

• In Java, a program consists of a collection of class definitions – each source code file contains the definition of the attributes and actions of a single class

• Unlike some other languages, the source code of a Java program cannot directly be run

• Instead, at runtime, the user creates objects from the classes in the program and instructs one of them to perform one of its permitted actions

• This initial object can then perform calculations, or create new objects and instruct them to perform actions etc.
BlueJ

• BlueJ is an integrated development environment (IDE) specially designed for teaching OOP
• Like any IDE, it permits the user to
  – Create and edit source code files
  – Compile the source code files into bytecode
• Unlike any other IDE, it also permits the user to
  – Interactively create objects from any class
  – Interactively invoke the actions of any object
  – Pass data to, and receive data from, any object

BlueJ allows the user to participate in the running program
Why is this important?

- Programming in an object-oriented language is like being a movie scriptwriter.
- A program – that is, a collection of source code files, is like a script in that it describes who the actors are and how they should behave.
- The actual execution of the program (that is, at runtime) is like the filming of the movie:
  - The objects are the “actors”
  - Each object behaves according to its own script
  - Any errors in the “script” will become apparent!
- Programming is even harder than scriptwriting because the programmer must specify everything exactly.
The BlueJ window

Class diagram

Object workbench
Compiling

- There is just one class called `BankAccount`
- The icon is shaded if the source code has not been compiled
- To compile it, press the “Compile” button
Syntax Error

If compilation fails, then BlueJ shows the line in source code that caused the error, and the error message returned by the compiler.
Creating Objects

Right clicking on the class icon brings up a menu – any line that starts with `new` can be used to construct an object of that class.

Objects in Java are created using the keyword `new`.
Construction

To construct the object, you must enter a *name* for that object, and any *initial data* that is used in constructing the object – often this will be the values of some or all of the object’s attributes.
Specifying the initial values

BlueJ tells the user what type of data (number, letter, string etc) is needed
Our first object

Objects appear on the *object workbench*
Create several objects

This program contains only one class

But we have made three objects, all instances of this class
The BlueJ user can *inspect* an object and look at the values of its attributes - right click on the object and select the menu item *Inspect*.
• This object has three *attributes*, and they have the values that were specified when the object was constructed

• The other objects will have the same collection of attributes, but *different values* for them
The actions

• The same menu also shows us the actions that the object can perform – in Java, these are called *methods*.

• In this case the three methods `deposit`, `getBalance` and `withdraw` are available.
Source Code

- The source code must specify *everything* about how each of the objects of that class behave
- It specifies the list of *attributes* that each object has
  - Each object in a class has the *same* attributes, but the *values* of those attributes may be different for different objects – each student has *some* student number, but different students have different student numbers
- It specifies how the objects are *constructed*
- It specifies what *methods* the objects have, and exactly what each method does
  - Each object in a class has the *same* collection of methods
When writing Java source code for a class, it is always necessary to consider three things:

1. What are the attributes of each object?
2. How are objects constructed?
3. What methods do the objects have, and how do they work?
CITS 1200 Java Programming

Methods
Scope of this lecture

• Interaction
• Accessors
• Other methods
• Existing classes - SimpleCanvas
Methods

• The *actions* that objects of a given class can perform are defined by the collection of *methods* of that class.

• We view the methods as being *services* that one object provides to another object.

• Whenever a method is *called*, or *invoked*, there are two objects involved:
  – The object that is *invoking* the method is called the *client*.
  – The object that is *running* the method – that is, providing the service is called the *server*.
Interaction via methods

• An important aspect of methods is that they form the primary mechanism by which objects communicate with each other
• When a method is invoked, the client object can send some information to the server object
• While the method is running, the server object performs calculations using its own attribute values and/or the values passed to it by the client
• When the method has completed, information can be returned from the server to the client
Methods in action

• We will start by considering how methods look from the client’s point of view – in other words we will call the methods of existing classes

• Next lecture, we will consider how the programmer must write the methods in order to achieve the desired behaviour

Peter, what is your student number?

The simplest type of method is called an accessor (sometimes called a getter) – this is when the client object simply wishes to access some information stored by the server object
In BlueJ

- There are two `BankAccount` objects on the workbench.
- The BlueJ user will play the role of the client, and call the methods of one or both of these objects.
Available methods

- Right clicking on the object brings up the list of methods for that object
- All objects of the same class must have the same collection of methods
An accessor

When the method `getBalance` is invoked...

...the result is returned to the client – in other words to the BlueJ user.
The method in action

getBalance();

The client

savings

1000000

balance

1000000

balance

1000000

1000000
A closer look

The method signature gives us some information about how the method will behave.

This is the type of information that will be returned from the server to the client.

This is the name of the method.

This is the type of information the client must supply to the server (in this case, nothing).


Beyond Accessors

• Accessors are very simple because they are only used to transfer information from one object to another

• The server object simply “reports back” to the client the value of one of its attributes

• The values of the attributes are unchanged after the method call – we say that the state of the object is unchanged

• Most interesting methods will do more than just examine the state of the object – they will actually change the state of the object
What does this method signature mean?

```java
void deposit(int)
```

A “void” will be returned – in other words this method does not return anything!

This is the type of information the client must supply to the server – in this case a single `int`

This is the name of the method
In BlueJ

The BlueJ user is prompted to supply the single `int` that is required – in this case the value 15000 has been entered into the textbox.
The method in action

```
deposit(15000);
```

The client

Nothing is returned!

`savings`

balance

100000

balance

1015000
Method call in Java

- The general Java syntax for a method call in a program is
  - The name of the target (server) object
  - A dot (full-stop, period)
  - The name of the method
  - Any arguments needed by the method
  - A semicolon

```java
savings.deposit(15000);
```

This line of Java code calls the `deposit` method of the object `savings`, with the argument 15000
BlueJ shows the equivalent underlying Java statement that the BlueJ user is actually making.
The role of the target

• What is the difference between
  ```java
  cheque.getBalance();
  savings.getBalance();
  ```
• The same method call is being made on *two different objects*
  – The first call is enquiring about the balance of the object `cheque`
  – The second call is enquiring about the balance of `savings`

```
peter.getStudentNumber();
```
Types of Method Calls

- In general, method calls may
  - Send information to the object, or not
  - Receive information from the object, or not

<table>
<thead>
<tr>
<th>Client sends information</th>
<th>Client does not send information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client receives information</td>
<td>getBalance</td>
</tr>
<tr>
<td>Client does not receive information</td>
<td>deposit withdraw</td>
</tr>
</tbody>
</table>
Real Life Analogies

- A method call that requests information, but does not send any information
  - “Peter, how old are you?”
  - `peter.getAge();`
  - Peter will respond with his age, which he can compute with no information from the client

- A method call that neither requests information, nor sends any information – it simply requests action
  - “Julia, please sit down”
  - `julia.sit();`
  - Julia will just sit down without saying anything
More real-life analogies

• A method call that both sends and requests information
  – Peter, what mark did you get for CS123?
  – `peter.getMark("CS123");`
  – Peter will respond with the details about his mark for CS123. Notice that Peter cannot answer the question sensibly without the information – CS123 – passed in the argument. It would make no sense to say “Peter, what mark did you get?”

• A method call that passes a special kind of information.
  – Peter, are you taller than James?
  – `peter.isTallerThan(james);`
  – The argument to this method call is the identity of another object
  – Peter will now need to make another method call, `james.getHeight()`, in order to answer this sensibly
Method Signature

- The *method signature* tells us whether information is to be sent, received or both

<table>
<thead>
<tr>
<th>What is returned</th>
<th>Name of method</th>
<th>What is sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>deposit</td>
<td>int</td>
</tr>
<tr>
<td>void</td>
<td>withdraw</td>
<td>int</td>
</tr>
<tr>
<td>int</td>
<td>getBalance</td>
<td></td>
</tr>
</tbody>
</table>

The method signature is the

- *Return type* of the method
- *Name* of the method
- *Types* (and order) of the arguments
Re-use

- One of the main aims of OOP is software re-use
- A class deals with a single logical concept, and provides mechanisms for storing any relevant data (the attributes) and mechanisms for accessing and manipulating that data (the methods)
- Therefore, in principle, a class should be able to be re-used in several different programs – it should form a software component analogous to the hardware components that make manufacturing such a success
- The Java language comes with a very extensive library consisting of thousands of existing classes
Using existing classes

• In order to use an existing class in your programs, you need to know how to get hold of the class, how to construct objects of that class, what methods it has, and what they do

• Java has a special documentation mechanism called javadoc that enables the programmers of a class to incorporate documentation directly into the source code, and have it translated automatically into HTML suitable for browsing

• BlueJ also uses this whenever using a class that has been suitably documented
The SimpleCanvas

• We will use a class called SimpleCanvas which is a modification of a class provided by the BlueJ team
• This class provides a simple mechanism for producing graphical output on the computer screen
• It is not part of the standard Java class libraries, but it has been carefully designed to behave in a very similar fashion to the much more complex classes in the Java graphics libraries
Constructing a SimpleCanvas

- Three constructors – one of them has no arguments, and just constructs a SimpleCanvas with the default size of 400x400 pixels
- Different sized SimpleCanvases can be constructed with the other constructors
A SimpleCanvas is created
Methods of SimpleCanvas

- The two fundamental methods of the SimpleCanvas are

  ```java
  void drawLine(int x1, int y1, int x2, int y2)
  void drawString(String text, int x, int y)
  ```

- They are both `void`, so they are not used to obtain information, but rather to request action – in this case the action is to draw on the SimpleCanvas window
  - `drawLine` draws a line from position \((x_1, y_1)\) to \((x_2, y_2)\)
  - `drawString` draws a String in position \((x, y)\)
Java’s graphics co-ordinates have (0,0) at the top-left of the window, with the first coordinate increasing left-to-right, and the second increasing top-to-bottom.
These examples look different because they are taken using BlueJ on Mac OS X - the differences are purely cosmetic.
## SimpleCanvas (drawing) - Microsoft Internet Explorer

All Classes
- Animator
- SimpleCanvas

```
SimpleCanvas(java.lang.String title, int width, int height)
  Creates and displays a SimpleCanvas with a white background and with automatic repainting after drawing commands.

SimpleCanvas(java.lang.String title, int width, int height, boolean autoRepaint)
  Creates and displays a SimpleCanvas of the specified size with a white background.
```

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drawLine</code></td>
<td>Draws a line on the SimpleCanvas between two points</td>
</tr>
<tr>
<td><code>drawString</code></td>
<td>Draws the specified String at the specified location on the SimpleCanvas</td>
</tr>
<tr>
<td><code>getFont()</code></td>
<td>Gets the font currently used for String drawing on this Canvas</td>
</tr>
<tr>
<td><code>getForegroundColour()</code></td>
<td>Gets the colour currently used for drawing on this SimpleCanvas</td>
</tr>
<tr>
<td><code>repaint()</code></td>
<td>If this SimpleCanvas does not automatically repaint after each drawing command, then this method can be used to cause a manual repaint</td>
</tr>
<tr>
<td><code>setAutoRepaint(boolean autoRepaint)</code></td>
<td>Sets the repaint mode to either manual or automatic</td>
</tr>
<tr>
<td><code>setFont(java.awt.Font newFont)</code></td>
<td>Changes the font for subsequent String drawing on this SimpleCanvas</td>
</tr>
<tr>
<td><code>setForegroundColour(java.awt.Color newColour)</code></td>
<td>Changes the colour for subsequent drawing on this SimpleCanvas</td>
</tr>
<tr>
<td><code>wait(int millis)</code></td>
<td>Causes execution to pause for the specified amount of time</td>
</tr>
</tbody>
</table>
CITS 1200 Java Programming

Class Structure
Scope of this lecture

• Structure of source code
• Defining Attributes
• Defining Constructors
• Defining Methods
Source code

• We will now consider how a programmer will write Java code in order to make the objects behave as we have seen they can.

• Recall that the programmer must specify precisely
  – The attributes of the objects of the class
  – How the objects of that class are to be constructed
  – How the methods of the objects behave

• We will focus in detail on the BankAccount class because this introduces the key concepts in a simple situation.
Structure of source code

- A Java source code file has a fixed structure

```java
public class ClassName {
    <definition of attributes>
    <definition of constructors>
    <definition of methods>
}
```

The entire definition is between the opening and closing curly braces

© CSSE, Gordon Royle 2008
Comments

• In addition to the “working” parts of the source code, the programmer can add comments that are ignored by the compiler, but can be used to document the code

• There are three styles of comment

    // a single-line comment after a double-slash

    /* a multi-line comment between an opening slash-star and a closing star-slash */

    /** special javadoc comments that can be automatically translated into HTML documentation */
Attributes

• The attributes of an object are represented by variables
• Each variable has a name which is used to symbolically refer to the value – it is called variable because this value may change as the program runs
• For example, a bank account uses a variable to store the balance of the account – this value varies when money is deposited to, or withdrawn from the account

266
balance

We can think of a variable as a “shoebox” containing a single value – the value can be changed, but it can only store one value
Variables in memory

• In memory, a fixed amount of memory space will be allocated for the variable

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

• For a numeric variable such as `balance`, the memory holds the binary representation of that number – in this case $266 = 256 + 8 + 2$. 

© CSSE, Gordon Royle 2008
Variable declarations

• A variable declaration gives the name of the variable and the type of the variable, followed by a semicolon
  int balance;
• The name of the variable is balance and the type of value that it can store is an int
  – An int can store an integer (whole number) and occupies 4 bytes
• An access modifier indicates which other objects can access this variable
  public int balance;
  private int balance;

The attributes of the objects of a class are defined by giving a variable declaration for each variable
public class BankAccount {
    private int balance;
    private int accountNumber;
    private String accountName;
}

Every object of the class BankAccount
has these three variables
Instance Variables

• Each *instance* of the class `BankAccount` has its *own* variable called `balance`, so these variables are called *instance variables*.
Avoiding confusion

• If there are lots of variables called `balance`, then how do we avoid confusing them?
• Each instance variable belongs to a *specific* instance
• We can distinguish them by using the name of the instance:
  savings.balance means
  “the balance variable belonging to the object savings”
  cheque.balance means
  “the balance variable belonging to the object cheque”
• The runtime environment keeps the variables belonging to different objects separate
Access modifiers

• Access modifiers do not change the type of a variable, or the name of a variable, but merely whether other objects can access it directly or not

• There are four access modifiers
  – public
  – private
  – protected

• We will only use two of them
  – public means that the variable is accessible by every object
  – private means that the variable is accessible only to objects of the same class

Access within a package is the default when no access modifier is used
• The Object Inspector allows the BlueJ user to look at the instance variables of any object on the object workbench
• The term “object field” is just another way of saying “instance variable”
Constructors

- The second thing that must be specified in the class definition is how objects of that class are *constructed*.
- A constructor has the same name as the class with a list of arguments that must be specified by the client.
- The code for the constructor then follows, again delimited by curly braces.
- Often, a constructor simply has to “set up” the initial values of the object according to the client’s specifications.
public BankAccount(String accName, int accNumber, int initBalance) {
    accountName = accName;
    accountNumber = accNumber;
    balance = initBalance;
}

• It is vital to understand what this means - the code
  (String accName, int accNumber, int initBalance)
specifies what information the client must *supply* when calling this method.
• The remainder of the code, in the curly brackets, specifies what should happen to this information during the construction of the object
Process of construction

• It is important to understand the process that occurs when an object is constructed
• Suppose a client calls
  
  new BankAccount("Bill",1234,1000);
• First the computer creates a new object, assigning space for the instance variables
• Secondly, the code in the constructor is run – the values given by the client are sent to the object in temporary variables called accName, accNumber and initBalance
• The code in the constructor just copies these values into the instance variables of the newly created object
new BankAccount("Bill",1234,1000);

• First, the runtime environment (ie the computer) creates the new object in memory, including space for each instance variable
• Then it runs the code in the constructor and does exactly what it says
Construction Process

The information supplied by the client is sent to the new object
Construction Process

• The code in the constructor is run - it specifies that the client-supplied information should be copied into the instance variables

• The temporary variables are then destroyed

• The object is now created and “ready to go”
Methods

• Now we must write code for the *methods* of the class, so that the objects can behave as their interface promises
• Start with the method `getBalance`
• What does this do?
  – If a client calls the `getBalance` method on a `BankAccount` object then the client wishes to know the balance of that account
  – the Java keyword `return` is used to return information to a client
  – As soon as a return statement is executed, the specified value is *immediately returned* to the client, and the method finishes

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

Methods can have access modifiers just like instance variables – this indicates which other objects are permitted to call the method
Method dissection

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

- `public` means that any other objects can call this method.
- `int` means that the method returns an `int`.
- `getBalance` is the name of the method.
- `return` is used to return information to the client.
- `balance` is the information that is returned.
The method in action

Client object makes the call
`savings.getBalance();`
The other methods

• What should the method deposit do?
  – It should add some money to the balance
  – The amount of money to be added is an argument of the method
  – The method does not return anything so is declared to be void

```java
public void deposit(int amount) {
    balance = balance + amount;
}
```
Method dissection

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

- The amount to be added is stored in a local variable called `amount`, which is created when the method is called, and destroyed when it is finished.

- The second statement is an assignment statement which says to take the values of `balance` and `amount`, add them together and store the result in `balance` – this overwrites the value previously stored in `balance`.
The method in action

Client object makes the call
savings.deposit(10);
After the method call

Nothing is returned to the client, but the state of the object `savings` has been changed.
Reflection on methods

• Consider again the deposit method
  
  ```java
  public void deposit(int amount) {
    balance = balance + amount;
  }
  ```

• The code refers to two variables – `balance` and `amount`

• It is critical to grasp that a method is always run by some target object, and that `balance` refers to “the balance variable of the object running this method”

• In Java, it makes no sense to just have a method call like
  ```java
  deposit(100);
  ```
  without specifying which object should run the method
Methods as services

• Another point that is important to grasp is that the main purpose of the methods of a class is to *provide services* for a client, not necessarily to directly solve a problem

• In general, the programmer does not know *which* methods will be called, *what order* they will be called in, or even *if* a particular method will *ever* be called

• This requires a “passive” mindset where the main consideration is “*What should happen if this method is called?*” rather than the traditional “active” mindset needed for a procedural program
Order is not important

- A source code file consists of the declaration of instance variables, the definition of constructors and the definition of methods
- These can occur in any order, providing each one is completely separate from each other one – having methods within methods is not permitted in Java
- It is traditional to have the instance variables together at the beginning, then the constructors, and then the methods
  - David Barnes tends to put the instance variables at the bottom of the class definition
Adding functionality

• A programmer is always free to add functionality to a class, simply by adding additional instance variables, constructors or methods.

• If a class has been “released” (as a library or as part of another program), then the behaviour of the existing methods should not be altered, but the implementation can be altered or improved if necessary.
CITS 1200 Java Programming

A Discipline of Programming
(with apologies to E.W. Dijkstra)

An Introduction to Programming by Contract and
UWA Java Tools for BlueJ
Complexity

The Boeing-777 commercial aircraft is a mix of proven equipment, many new technologies and some new features. Altogether the digital aircraft contains over $5 \times 10^6$ lines of code.

CITS1200 projects are approx

- 250 lines of code and require around 25 person hours of effort
- So the B-777 code is the size of **20,000** CITS1200 projects taking **57** person years
- Writing real code requires discipline and standards to ensure that your code interfaces correctly with the whole system
Lecture Outline

- Programming in the large
- Java syntax and the Java compiler
- Public interface of a class
- Method signatures
- Private fields and methods
- How to read Javadoc documentation
- How to read a JUnit4 test cases
- Checkstyle: see lab sheets and http://www.csse.uwa.edu.au/UWAJavaTools/checkstyle/
UWA Java Tools

• The standard release of BlueJ runs only the outdated JUnit3 system
• Patrick Doran-Wu UWA has developed a JUnit4 version in consultation with the BlueJ team at Kent.
• We will be using his BlueJ 99.0.1junit4 in this unit
• BlueJ with JUnit4 is already installed in the CSSE labs

• You can download it (for home) from


• This is a new release, so please email rachel with any bugs discovered
Java Compiler

Programs must conform to Java syntax in order to be executable.
Some of the most common errors are forgetting a semi-colon or a bracket.
Use Edit/Auto-layout to make it easier to spot these errors.
Missing semi-colon syntax error
Missing bracket syntax error
Public Interface of a Class

```java
public class BankAccount {

    private int balance;
    private String accountName;
    private int valueDeposits;
    private int valueWithdrawals;
    private int maximumBalance;
    private int minimumBalance;

    /**
     * construct a bank account with given initial balance
     * @param accountName unique identifying string
     * @param balance opening balance of account, assumed non-negative
     */
    public BankAccount(String accountName, int balance) {
        this.accountName = accountName;
        this.balance = balance;
        this.valueDeposits = balance;
        this.valueWithdrawals = 0;
        this.maximumBalance = balance;
        this.minimumBalance = balance;
    }

    public int getBalance() {
        return balance;
    }

    public String getAccountName() {
    }
}
```
Javadoc view of the class

Class BankAccount

```java
java.lang.Object
  BankAccount
```

public class BankAccount extends java.lang.Object

Class BankAccount - a simple model of a bank account designed to illustrate the concepts of classes, objects, and methods

Version:
  Feb 2011

Author:
  Rachel Cardell-Oliver based on version by Gordon Royle

Constructor Summary

```java
BankAccount(java.lang.String accountName, int balance)
  construct a bank account with given initial balance
```

Method Summary

```java
void applyInterest(double rate)
  calculate and update interest under the condition that account balance is negative and rate is positive

void deposit(int amount)
  deposit amount by adding to balance
```
Method Signatures

/**
 * construct a bank account with given initial balance
 * @param accountName unique identifying string
 * @param balance opening balance of account, assumed to be non-negative
 */
public BankAccount(String accountName, int balance)

Your implementation of all public methods must match the specification exactly – that is the contract that allows Java programs to be built from collections of classes
**More Method Signatures**

/**
 * deposit amount by adding to balance
 * @param amount must be non-negative
 */

public void deposit (int amount) ...

/**
 * withdraw amount by decreasing balance
 * @param amount must be non-negative
 */

public void withdraw (int amount) ...
Private Fields and Methods

• Java classes operate a “need to know” policy
• They offer a public signature of services that other classes can use.
• Everything else is hidden to avoid unintended alterations.
• For example, if a bank account balance field is public, then any other class can alter the balance.
• Fields are always hidden by using the `private` key word
• Methods that are only required within a class, and not offered as a service to other classes, are also hidden
• These private methods are called `helpers`
• Exercise: look for all the private declarations in sample code.
How do you know what to implement?

• A method signature tells you the name, return type and parameters.
• Javadoc gives a prose description of the intended purpose.
• But how do you know exactly what the method should do?
• For example, if the amount parameter of the deposit method is -100, what should the resulting balance be? Isn’t that a withdrawal rather than a deposit?

• Answer: JUnit test cases specify the intended behaviours of a method by giving examples of the expected result for each of the different cases of the method.
How to read JUnit test cases

```java
/**
 * deposit method increases balance by given amount
 */
@Test public void testBalanceAfterDeposit() {
    normalAccount.deposit(200);
    assertEquals(1200, normalAccount.getBalance());
}

/**
 * deposit has no effect if amount is negative
 */
@Test public void testBalanceAfterNegDeposit() {
    normalAccount.deposit(-200);
    assertEquals(1000, normalAccount.getBalance());
}
```
When your code fails a test read the test code to see what is missing
What’s in a test case?

Unit test cases have the following form:

1. describe the method’s inputs
2. describe the expected outputs
3. call the method and observe actual output
4. compare expected with actual and report:

   assertEquals( comment, expected, actual);
Before every test case

```java
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

/**
 * The test class BankAccountBasicTest.
 * test for basic functionality of the BankAccount class
 * @author Rachel Cardell-Oliver
 * @version February 2011
 */
public class BankAccountBasicTest
{
    private BankAccount normalAccount, emptyAccount;

    /**
     * set up sample accounts to use for the tests
     */
    @Before public void setUp() {
        normalAccount = new BankAccount("John", 1000);
        emptyAccount = new BankAccount("Jane", 0);
    }
}
```
How a JUnit test suite is executed

1. `setUp` method is executed
2. the first test method in the class is executed
3. `setUp` method is executed
4. the second test method in the class is executed

Continue until all test cases are executed.

**IMPORTANT** each test case is *independent* of previous tests
How Test Cases are Chosen

• Typical cases
  – Sanity check that method code works as expected, but try to choose tricky cases anyway!

• Special cases and Boundary cases
  – Most errors occur for boundary values, eg empty or full array, -1,0,1 etc.

• Exceptional cases
  – Illegal input, divide by 0, un-initialized parameters
Heads up: method signature errors

Solution: check the deposit method in your class, it probably has the wrong signature e.g. `makeDeposit` or wrong parameter list.
Coding Standards with Checkstyle

A goal of any software developer should be to write consistently clear, high quality, maintainable code. This is not always easy and requires a certain amount of discipline at the best of times. One way to help achieve high quality code is by using coding standards.


- CS-naming
- CS-blocks
- CS-coding
- CS-hiding
- CS-complexity
- CS-javadoc

© CSSE, Rachel Cardell-Oliver 2011
Summary

In CITS1200 you will be programming by contract in all your lab exercises and projects. In order to do this, you need to take account of information provided by all of the following:

- Java compiler
- Class and method signatures
- Javadoc descriptions of methods
- JUnit test cases for methods
- Checkstyle coding standards
  - (the minimal UWA Java Tools ones)
CITS 1200 Java Programming

Types
Scope of this lecture

• Types in Java
  – the eight primitive types
  – the unlimited number of object types
• Values and References
• The Golden Rule
Primitive types

• Every piece of data that is stored or moved around in Java must have a type
• Java supplies eight primitive types for the most fundamental pieces of data

short
int
long
float
double
double
boolean
char

© CSSE, Gordon Royle 2008
Integers primitive types

- Four of the types are used for storing whole numbers
- Each occupies a fixed amount of memory
  - `byte` uses 1 byte and stores integers from $-128$ to $127$
  - `short` uses 2 bytes and stores integers from $-32768$ to $32767$
  - `int` uses 4 bytes and stores integers from $-2147483648$ to $2147483647$
  - `long` uses 8 bytes and stores integers from $-9223372036854775808$ to $9223372036854775807$
    (that is $-2^{63}$ to $2^{63} - 1$)
Floating point primitive types

- There are two types used for floating point values – in other words, numbers with decimal points
  - `float` uses 4 bytes and stores floating point numbers, positive or negative, in the range $1.4\times 10^{-45}$ to $3.4028235\times 10^{38}$
  - `double` uses 8 bytes and stores floating point numbers, positive or negative, in the range $4.9\times 10^{-324}$ to $1.7976931348623157\times 10^{308}$
Declaring variables

- In Java, all variables must be declared before they are used
- The declaration specifies the *type* and *name* of the variable
- When a primitive variable is created, a fixed amount of memory is set aside for that variable
  - `byte b;`
  - `double temp;`
  - `int balance;`
Assignments to numeric types

• An assignment sets the value of a variable
  ```java
  int x; // this is a declaration
  x = 100; // this is an assignment
  ```

• Here are some legal assignments
  ```java
  int x;
  double y;
  double z;

  x = 200;
  y = -354.278;
  z = 23e5; // this is scientific notation
  ```
Typecasting

- The compiler will warn you if you attempt to use a “smaller-sized” variable than it thinks the value needs.
- For example, if you attempt
  ```java
  float b;
b = 12.35;
  ```
  then the compiler will warn you of a “possible loss of precision” because 12.35 is regarded as a `double`.
- If this is really what you want, then you can cast the value
  ```java
  float b;
b = (float) 12.35;
  ```
  – for literals, there is a “quick cast” – `b = 12.35f;`
Non-numeric primitive types

- boolean is used to hold one of the two boolean values true or false
  
  ```java
  boolean isDone; // declaration
  isDone = false; // assignment
  ```

- char uses two bytes and holds a Unicode character
  
  ```java
  char ch; // declaration
  ch = 'A'; // assignment
  ```
Literals

• Actual values that you type into the source code are called *literals* - these are things like

100, -23, true, 'Z', 37, 1e-5, 22, 3.1415926

• *Everything* in Java has a type, even the literals, so the types of these are respectively

int, int, boolean, char, int, double, int, double
Initialization

• If a variable is given an initial value directly after it is declared, then you can combine the two steps into one
  ```java
  int x;
  x = 10;
  boolean isFinished;
  isFinished = true;
  ```

  is equivalent to
  ```java
  int x = 10;
  boolean isFinished = true;
  ```
Type-checking

• The Java compiler does extensive type-checking to make sure that the program does not have any obvious problems.
• The compiler will not allow the wrong type of value to be assigned to a variable:
  ```java
  int x;
  x = true;   // cannot assign boolean to int
  ```
• Similarly, the compiler will not allow the wrong type of value to be used as a method or constructor argument.
• What about this?
  ```java
  int x = 20;
  double y;
  y = x;
  ```
  Although the types don’t match perfectly, the computer knows that there is a value of type double corresponding to any value of type int – in this case the value 20.0 is assigned to y.
Primitive types in assignments

• If a primitive type appears on the left-hand side of an assignment,
  
  ```java
  int x;
  x = 37;
  x = 10;
  ```

  then the value in the shoebox gets replaced.

  The new value *overwrites* the old
Primitive types in expressions

• If a primitive type appears on the right-hand side of an assignment,
  
  ```java
  int y;
  y = x;
  ```

  then the value in the shoebox `x` is copied into `y`

```
x = 00101010000000000000001101010
y = 0101010100000000000000101010
```

```
x = 00101010000000000000001101010
y = 0101010100000000000000101010
```


**Primitive types as arguments**

- If a primitive type is used as the argument to a method, then its *value* is copied into a temporary variable accessible *only* from within the method
  
  ```java
  int x = 100;
  savings.deposit(x);
  ```

- What happens here is:
  - a variable called `x` is created, and given the value 100
  - the `deposit` method expects an argument called `amount` so a *temporary local variable* `amount` is created, and the value of `x` is copied into it
  - the variable `amount` is destroyed when the method finishes
  - the variable `x` still exists, and still contains the value 100
Reference types

• Everything that is not a primitive type is a reference type
• Every class defines a type!
• There are classes that are automatically available, library classes for which you must specify the “full name” and user-created classes
  – String is a library class that is automatically available
    • String is the “short name” for java.lang.String
    • Any class from the package java.lang can be used directly
  – java.awt.Color is a library class
    • You must use the full name for this class
  – BankAccount and SimpleCanvas are user-created classes

© CSSE, Gordon Royle 2008
Variable declarations

• Declaring reference types is just the same as for primitive types, with the type occurring before the name
  
  String s;
  BankAccount b1;
  BankAccount b2;
  SimpleCanvas sc;

• This has created the variables – the “names” for the objects – but we have not actually created the objects themselves

Creating objects is (almost always) done with the Java keyword new
Creating objects

- To actually create the object, the constructor is called
  
  \[
  \text{sc} = \text{new SimpleCanvas();}
  \]
  
  \[
  \text{bl} = \text{new BankAccount(“BillG”,12345,10000);}\]

- An object is fundamentally different from a primitive value
  - It does not have a fixed size when it is created
  - Its size can change during runtime

- For this reason, objects are stored in a different part of the computer’s memory, and the value that is stored in the variable is a reference to the object
  - A reference contains the information needed to find the actual object in the computer’s memory
  - The variable refers to an object (rather than contains an object)
The variable $b_1$ contains a reference to some location in the heap.

The actual object that $b_1$ refers to is stored on the heap.

Some other object.

Garbage

© CSSE, Gordon Royle 2008
Another viewpoint

- Another way to visualize this is to consider that an object is *too big* to fit in a shoebox, so instead the shoebox contains a *reference* to the object.

- If somebody sends you a postcard, it is put directly into your mailbox, but if someone sends you a *package* then the postman puts a card into your mailbox telling you *where to go* to pick up the package.
  - The package is the *object*, the card is the *reference*.
The crucial difference

- A primitive variable directly holds an actual value
- A reference variable holds instructions about how to locate an object in the heap
The Golden Rule

• The behaviour of reference types and primitive types appears quite different, but it all hinges on the following observation
  – A variable of primitive type contains a value
  – A variable of reference type refers to an object

The Golden Rule

Whenever a variable is used, it is the contents of the shoebox that is used

• this is the value for a variable of primitive type
• this is the reference for a variable of reference type
Reference types in assignments

- If we use a variable of reference type in an assignment, then only the reference is assigned.
- Consider the following statements:

```java
BankAccount b1;
BankAccount b2;
b1 = new BankAccount("Bill Gates", 1234, 100);
b2 = b1;
```

![Diagram showing reference types in assignments]
Manipulating reference types

• What happens if we do

```java
b1.deposit(200);
```
Two references, one object

• And follow up with

```java
b2.deposit(500);
```

b1 and b2 are different references to the same object!
The memory viewpoint

As \( b_1 \) and \( b_2 \) contain the same “location information” they refer to the same object!
Question

• What is the effect of the following statements?

```java
BankAccount b1;
BankAccount b2;
b1 = new BankAccount("Bill Gates", 1234, 100);
b2.deposit(200);
```

• To answer this, we need to work through the statements one at a time, considering what each of them does
Answer

• The variable *declarations* create two variables that will hold references
  
  ```java
  BankAccount b1;
  BankAccount b2;
  ```

• The two references are set equal to the special value `null` because there are not any objects for them to refer to!
  
  ```java
  b1 = new BankAccount("Bill Gates", 1234, 100);
  ```
Null Pointer Exception

- The next statement
  \[ b2.\text{deposit}(100); \]
  attempts to deposit 100 to the object to which \( b2 \) refers

- But \( b2 \) is not storing a reference to \textit{any object} at all
  - Therefore the program will cease and produce the error message \texttt{NullPointerException}
  - You will see many, many \texttt{NullPointerExceptions} in your programming career!
Enumerated Types

- Java 1.5 (also known as Java 5.0) has added a new feature known as an *enumerated type*
- An enumerated type is a special class used to represent a *fixed finite collection* of values
- For example, suppose you are writing a program for some card-playing game (bridge, poker, solitaire etc)

```java
public class Card {
    private <what here?> value;
    private <what here?> suit;
}
```
• We need variables to represent the value and the suit of the card, but what type should they be?

• The usual solution to this is to just use something like an int and to have some convention about what the values mean

```java
public class Card {
    private int value;  // Use 1 for Ace, 2 for 2, etc
    private int suit;   // 0=club, 1=diamond, 2=heart, 3=spade

    public Card (int v, int s) {
        value = v;
        suit = s;
    }
}
```
Problems

• This solution has problems, because everyone who uses the class must know and remember these codes

• There is no error checking if the user accidentally confuses the two parameters - does the following statement construct the Ace of Hearts or the Two of Diamonds?

\[
\text{Card } c = \text{new Card}(1,2);
\]
Enumerated Types

```java
public enum CardValue {
    ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN,
    JACK, QUEEN, KING
}

public enum CardSuit {
    CLUBS, DIAMONDS, HEARTS, SPADES
}

public class Card {
    public CardValue value;
    public CardSuit suit;

    public Card (CardValue v, CardSuit s) {
        value = v;
        suit = s;
    }
}
```

© CSSE, Gordon Royle 2008
Using enumerated types

• Now the user *must* use a legal value when creating the objects

```java
Card c = new Card(CardValue.ACE, CardSuit.HEARTS);
```

• The code is readable and properly *type-checked*.

• Java’s enumerated types are very sophisticated and this example just shows the simplest possible use - in fact, an `enum` type is a fully-fledged class and so can have methods.
CITS 1200 Java Programming

Using Other Classes
Scope of this lecture

• Composition and Delegation
• this
• Clients, Servers and Cooperation
Composition

- Composition is a powerful and fundamental object-oriented programming technique.
- It is a technique where a programmer creates a class that uses the functionality and facilities of other classes (preferably a library class or other existing class).
- This is achieved by the class having instance variables referring to objects of the other classes.
- Using composition, a sophisticated class can often be developed quickly with only moderate programming effort because the programmer is freed from “reinventing the wheel” by using existing classes.
EtchASketch

• We will devise a class to represent a computer version of the classic children’s toy “Etch A Sketch”
• This toy has a rectangular drawing area and two buttons
EtchASketch

• There is always a current “drawing point” and rotating the buttons causes the drawing point to move up, down, left or right leaving a visible trail behind it that constitutes the drawing

Etch A Sketch is a registered trademark of The Ohio Art Company
Class Design

• We must decide on what the EtchASketch object can do - what services it provides to the client

• A real Etch A Sketch object performs as follows
  – It draws vertical or horizontal lines from the current drawing point according to the user’s rotation of the knobs, and moves the drawing point correspondingly

• Therefore our class will have the following methods

  void up(int)
  void down(int)
  void left(int)
  void right(int)

Each method takes an integer distance representing a number of pixels, and draws a line of that length
Class Design

- We know of an existing class - SimpleCanvas - that can provide line drawing methods, so we will build the EtchASketch class by using this.

- Each EtchASketch object will be composed of
  - One SimpleCanvas object, and
  - Instance variables to store the location of the drawing point.

- When the EtchASketch user makes method calls to cause some drawing action to happen, the EtchASketch object will use the SimpleCanvas object to actually do the drawing.
Instance Variables

public class EtchASketch {
    private SimpleCanvas sc;
    private int xPos;
    private int yPos;
}

• Each object has three instance variables - one SimpleCanvas to do the drawing, and two ints to represent the coordinates of the drawing point.
• They are all declared to be private because the client never needs to access them directly
In BlueJ

The arrow indicates that the `EtchASketch` class *USES* the `SimpleCanvas` class.

BlueJ’s class diagram shows the relationships between classes in each project.
Constructor

- The role of the constructor is to “set up” the object so that it is ready to receive and process method calls.
- In this case, the constructor must:
  - Create a blank `SimpleCanvas` object,
  - Initialize the drawing point to some position.
- The programmer must make some decisions:
  - Should the client have any control over the size of the drawing area, or should the programmer choose it?
  - Where should the initial drawing point be located?
As simple as possible

- The simplest solution is to allow the client no control, but to just create the default sized 400 x 400 SimpleCanvas and put the initial drawing point in the middle
  - At this point we are just intending to reproduce the drawing area of the Etch A Sketch and not the coloured plastic casing or knobs!

```java
public EtchASketch() {
    sc = new SimpleCanvas();
    xPos = 200;
    yPos = 200;
}
```
What happens on construction

• The client calls
  
  \texttt{new EtchASketch()}

• The computer creates the new object, which then calls
  
  \texttt{new SimpleCanvas()}
  
  – So the client (user) creates the EtchASketch object, and the EtchASketch object creates the SimpleCanvas

• Then the value 200 is assigned to each of the variables $xPos$ and $yPos$
Right-clicking on the `EtchASketch` class brings up the menu for constructors - there is only one defined and so we must choose that one - it requires no information from the client, the client is just asked for the name of the new object, which can freely be chosen.
Inspect the new object
The squiggly arrow

- For the variables of primitive type, the Object Inspector directly reports on their *value*, but for any variables of reference type the Object Inspector just shows a squiggly arrow!

- The squiggly arrow is a visual reminder that the variable `sc` is a *reference* to an object of type `SimpleCanvas`
After construction

A SimpleCanvas, whose internal details are irrelevant to us
The methods

- The methods translate the client’s requests such as `etch1.up(25)` into commands for the `SimpleCanvas`

```java
public void up(int distance) {
    sc.drawLine(xPos, yPos, xPos, yPos - distance);
}
```

- If the current drawing point is at `(xPos, yPos)` and we want to draw a line that goes *up* by an amount `distance`, then the end of that line will be `(xPos, yPos - distance)`

- This method *passes on* the drawing request to the `SimpleCanvas` to get the line drawn, or in other words, it *delegates* some of its work
Co-operation

- An object provides a more complex service by co-operating with objects from other classes.
- Much of the complexity is hidden because most of the work is delegated to another object.

```
Client -> etch1 -> sc

EtchASketch  SimpleCanvas

Client -> Pierre -> Joe

Waiter  Chef

up(50)  drawLine(200,200,200,150)

Bok-choy with black beans, please

One number 23

© CSSE, Gordon Royle 2008
```
Clients and Servers

• When the BlueJ user asks an `EtchASketch` object to draw a line
  – The BlueJ user is the `client`, and
  – The `EtchASketch` object is the `server`

• When the `EtchASketch` object fulfills this request by delegating the work to the `SimpleCanvas`
  – The `EtchASketch` object is the `client`, and
  – The `SimpleCanvas` object is the `server`

In a running program, an object can act as a client, a server, or more usually as both a client and a server
Whoops..

- This method will draw the line correctly, but it does not update the position of the drawing point - we need to update the instance variables to reflect the new drawing position at the end of the line

```java
public void up(int distance) {
    sc.drawLine(xPos, yPos, xPos, yPos-distance);
    yPos = yPos - distance;
}
```
The other methods

```java
public void down(int distance) {
    sc.drawLine(xPos, yPos, xPos, yPos+distance);
    yPos = yPos + distance;
}

public void left(int distance) {
    sc.drawLine(xPos, yPos, xPos-distance, yPos);
    xPos = xPos - distance;
}

public void right(int distance) {
    sc.drawLine(xPos, yPos, xPos+distance, yPos);
    xPos = xPos + distance;
}
```
Improving *EtchASketch*

- The real Etch A Sketch is limited to just four drawing commands, but we can easily add extra ones to the computer version.
- Suppose we decide to add a command to draw a *square* with top-left corner at the current drawing point.
We can work out the lines that need to be drawn to make up a square and then delegate the actual drawing to the SimpleCanvas.
The square method

public void square(int side) {
    sc.drawLine(xPos,yPos,xPos+side,yPos);
    sc.drawLine(xPos+side,yPos,xPos+side,yPos+side);
    sc.drawLine(xPos+side,yPos+side,xPos,yPos+side);
    sc.drawLine(xPos,yPos+side,xPos,yPos);
}

• When a client calls the method square, it then makes four calls to the method drawLine

• There is no need to update xPos and yPos because their final values are the same as their original ones
Second Approach

• Another approach is to express the more complicated method \texttt{square} in terms of the \textit{existing} methods

  
  public void square(int side) {
    this.down(side);
    this.right(side);
    this.up(side);
    this.left(side);
  }

  
  This method is calling other methods of the \textit{same} object
The keyword this

- In Java, any object can access a reference to *itself* using the keyword this
- In other words, this is the Java equivalent of “me”
- Recall that a method is called by specifying:
  ```java
targetObject.methodName(arguments)
  ```
- So in the first approach to the square method, the method delegates work to the `SimpleCanvas`, with reference `sc`
  ```java
  sc.drawLine(...)
  ```
- In the second approach, the method calls another method of itself, and so it refers to itself with this:
  ```java
  this.up(...)
  ```
No target object

- It is important to understand how to use `this` because often you don’t need to!
- Whenever a method call appears with NO target object, the compiler will assume that the target object is `this`

```java
public void square(int side) {
    up(side);
    right(side);
    down(side);
    left(side);
}
```
No target object

• The compiler will silently behave as though every such method call begins with `this`

```java
public void square(int side) {
    this.up(side);
    this.right(side);
    this.down(side);
    this.left(side);
}
```
CITS 1200 Java Programming

Expressions
Scope of this lecture

• Simple expressions
• Compound expressions
• Arithmetic operators
• Relational and logical operators
Expressions

• An expression is a portion of Java code that is evaluated to yield a value
  – any expression has a type and a value

  total = savings.getBalance() + cheque.getBalance();

• Whenever a statement is executed, each expression will be evaluated and replaced with its value
  – The process of evaluating expressions and doing something with the resulting value is the fundamental process of computing
Evaluation of Expressions

• A simple example
  ```java
  int x;
  x = 10 + 15;
  ```
  The first line creates a variable called `x` of type int, while the second line will evaluate the expression `10+15`, replace it with its value (which is `25`) and then assign the resulting value to `x`

• Evaluation of an expression may take several steps and require the cooperation of several objects
  ```java
  double y;
  y = (b1.getBalance()*0.01) + (b2.getBalance()*0.05);
  ```
Types of Expression

• There are several types of expression
  – *literals* and *names*
  – *method invocations*
  – *object creations*
  – *compound expressions* built up with *operators*
    • *arithmetic* operators
    • *relational* operators
    • *logical* operators
**Literals**

- *Literals* are values that are “hard wired” into the code
  
  23, 10009, ‘a’, –23.456, 10e–5, 100e2

- In Java, the types `int` and `double` are dominant, in that any literal that does not use a decimal point is assumed to be of type `int`, and any literal uses a decimal point (or scientific notation) is assumed to be a `double`.

- The Java compiler will complain if you try to assign a value to a “smaller” type.

```java
float f;
f = 23.5;
```
Casting

• If the programmer really wants to assign a double value to a float variable, then the compiler can be prevented from complaining by using a cast

• The number is simply truncated to fit the new type

```java
float f;
f = (float) 23.5;
```

This says to the compiler “I know this looks dumb, but I do know what I’m doing so please just treat this number as a float rather than a double”
Names

- *Names* are the declared *variables* that are in scope; the expression has the value and the type currently associated with that variable
  - the *value* in the shoebox if it is a primitive type
  - the *reference* in the shoebox if it is a reference type

etch1, accountNumber, balance, sc, x
Method invocations

- Another form of expression is the calling of a non-void method of some object
  - The value of the expression is the returned value
  - The type of the expression is the return type of the method
- The method call (here b1 is a BankAccount)
  - b1.getBalance() is an expression
    - Its type is int, because the return type of the method is int
    - Its value is the current value of b1’s balance
- You can use such an expression anywhere that an int is allowed – the compiler will complain if you violate this
  - type checking is a valuable aid to debugging!
Object Creations

- The construction of an object with `new` returns a reference to the newly created object, so this is also an expression.
- The expression
  ```java
  new SimpleCanvas()
  ```
  returns a reference to a `SimpleCanvas` object and so can be used anywhere that such a reference is valid.
**Compound Expressions**

- A *compound expression* is obtained by combining one or more simple expressions with an *operator*
  
  \[ 5 + 3 \]

  is a compound expression, obtained by combining the literals 5 and 3 with the operator +
  
  - The *value* of this expression is 8
  - The *type* of this expression is `int`

- We have already used one compound expression
  
  - `balance` is a simple expression
  - `amount` is a simple expression
  - `balance + amount` is a compound expression
Classes of Operators

- Java has a large number of operators
- *Arithmetic* Operators
  - take numeric expressions and perform arithmetic on them, producing numeric answers
- *Relational* Operators
  - take numeric, or other, expressions and make logical tests of equality, inequality etc, producing *boolean* answers
- *Logical* Operators
  - take boolean expressions and combine them using the rules of logic, producing boolean answers
- *Miscellaneous* Operators
  - other operators, acting on *Strings*, bit patterns etc.
**Arithmetic Operators**

- We will concentrate on the *arithmetic* operators first
- They are as follows:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Operation</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+, −</td>
<td>Unary plus</td>
<td>Right-to-left</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unary minus</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*, /, %</td>
<td>Multiplication</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Division</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remainder</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>+, −</td>
<td>Addition</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subtraction</td>
<td></td>
</tr>
</tbody>
</table>
Association

• The association rules explain how to combine a sequence of operators of the same precedence
• Consider evaluating the expression
  \[100 / 10 / 5\]
• The / operator *associates* left-to-right so the value is
  \[(100 / 10) / 5\]
• If it associated right-to-left, it would be \[100 / (10 / 5)\]
• The association rules just follow normal mathematical usage, so there are no surprises
Arithmetic Operators

• These behave (almost) as you would expect

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 + 53</td>
<td>76</td>
</tr>
<tr>
<td>13 * 5</td>
<td>65</td>
</tr>
<tr>
<td>30 / 5</td>
<td>6</td>
</tr>
<tr>
<td>2.37 * 2</td>
<td>4.74</td>
</tr>
<tr>
<td>12 - 15</td>
<td>-3</td>
</tr>
</tbody>
</table>

• But what is $5 \times 6 + 3 \times 2 - 4 + 6 \times 11$?
Precedence

• The compiler uses the precedence and association rules to determine the order of evaluation
  \[ 5 \times 6 + 3 \times 2 - 4 + 6 \times 11 \]
  becomes
  \[ (5 \times 6) + (3 \times 2) - 4 + (6 \times 11) \]
  because * has a higher precedence than + or −

• The resulting expression is calculated *left to right*
  \[ ((30 + 6) - 4) + 66 \]

• The programmer can *explicitly* use parentheses if a particular order is required
Watch your types!

- Every expression has a type, which depends on the operators involved and the types of the operands.
- In particular, the division operator `/` returns an `int` if its operands are `ints`.
- So,
  - `7/5` yields 1
  - `12/4` yields 3, but so does `13/4` and `14/4`
  - `7.0/5` yields 1.4, as does `7/5.0` or `7.0/5.0` because the expression is a floating point number.
- The integer is obtained by truncating the expression.
  - `100/51` yields 1.
Integer Division

• Java’s integer division is *always* a fertile source of difficult-to-trace bugs
  
  ```java
  double d = 8 / 5;
  ```

• After this statement the value of d is 1.0 and *not* 1.6!

• Why?
  – *first* the expression 8 / 5 is evaluated, and as both the arguments are of type int, the answer is truncated to an int
  – *then* the value 1 is assigned to the double variable, and hence d has the value 1.0

• This leads to some potentially confusing situations
  
  2.0 / 3 * 6 is *not equal* to 2 / 3 * 6.0
The remainder operator

- The operator `%` returns the *remainder* in a division with two integer arguments

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 % 3</td>
<td>1</td>
</tr>
<tr>
<td>12 % 4</td>
<td>0</td>
</tr>
<tr>
<td>13 % 7 % 4</td>
<td>2</td>
</tr>
<tr>
<td>13 % (7 % 4)</td>
<td>1</td>
</tr>
</tbody>
</table>

- An important Java *idiom* or *cliché* is the use of `/` and `%` together
  - If someone is 7000 days old, then how old are they?
    - 7000 / 365 years, plus 7000 % 365 days
The CodePad

Turn on the CodePad by using the View menu and selecting Show Code Pad

This is the CodePad which allows you to enter either expressions or statements and have them performed immediately.
Evaluate an expression

• Enter any expression in order to have it immediately evaluated
• Here we discover that $15 \% 11$
  has the value 4 and the type `int`
• Notice that the CodePad knows you want to evaluate an expression because it has no semicolon
Relational operators

• Relational operators are crucial in allowing a program to choose among different courses of action depending on the results of calculations
• They are often used to determine a program’s flow of control, using if statements (and others)
  – “if the balance is less than zero, then print out an overdraft statement”

```java
if (balance < 0) {
    // print out the overdraft statement
}
```
Relational Operators

- The six relational operators compare numeric values and return a boolean

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
<td>Is a equal to b</td>
</tr>
<tr>
<td>a != b</td>
<td>Is a not equal to b</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>Is a greater than b</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>Is a less than b</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>Is a greater than or equal to b</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>Is a less than or equal to b</td>
</tr>
</tbody>
</table>
Simple relational expressions

2 == 3 is false
4 <= 5 is true
5 != 7 is true
2*6 == 12 is true
7.0/3.0 == 2.33333333333 is probably false

• A relational expression
  – has the type boolean
  – has value true or false, depending on the comparison

• It is quite legitimate to use relational expressions in assignments
  boolean isOverDrawn = (balance < 0);
The special case of equality

• Correct use of the relational operator == is critical
• With variables of primitive type, the only thing to worry about is using floating point numbers!
  
```java
  boolean test1 = (1 + 2 == 3);
  boolean test2 = (0.1 + 0.2 == 0.3);
```

• test1 is true (as it should be), but test2 is false!
  – Hmmm, what exactly is 0.1 + 0.2 if it is not equal to 0.3?

Round-off error is caused by truncating infinite decimal expansions to 15 or so decimal places
Equality for reference types

• The tricky bit is equality for reference types

BankAccount b1 = new BankAccount("Bill", 123, 1000);
BankAccount b2 = new BankAccount("Bill", 123, 1000);

• What is the value of the expression b1 == b2?

The Golden Rule

Whenever a variable is used, it is the contents of the shoebox that is used

• this is the value for a variable of primitive type
• this is the reference for a variable of reference type

© CSSE, Gordon Royle 2008
Equality uses the references

- Here, b1 and b2 are references to *two separate objects*
  - The objects have identical instance variables, with identical values, but they are not the *same* object!

Therefore, the expression

\[ b1 == b2 \]

is *false*!
The memory viewpoint

As b1 and b2 contain different “location information” they refer to different objects!
Logical Operators

- Boolean expressions can be combined into compound expressions using operators for NOT, AND and OR

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &amp;&amp; b</td>
<td>a AND b</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>!a</td>
<td>NOT a</td>
</tr>
</tbody>
</table>
Use of the logical operators

- The logical operators combine boolean expressions in the obvious way
- Suppose that `<bool1>` and `<bool2>` represent arbitrary boolean expressions

```java
<bool1> && <bool2>
    is true if both <bool1> and <bool2> are true

<bool1> || <bool2>
    is true if either <bool1> or <bool2> are true

! <bool1>
    is true if <bool1> is false, and false if <bool1> is true
```
Shortcut Evaluation

• Java uses *shortcut* evaluation of the logical operators
  – as soon as it knows the final answer it stops calculating

```java
if (balance > 2000000 || accountName == "Kerry Packer") {
    // send grovelling letter
}
```

• There are two relational tests, one on `balance` and the other on `accountName`, combined with an *OR*
  – if the first test is *true*, then the entire boolean expression will be *true* regardless of the second test
  – thus, Java saves time by not even bothering to do the second test
CITS 1200 Java Programming

Repetition
Scope of this lecture

- Repetition
- The for loop
- The while loop
Dynamic Behaviour

• The great power of computers, and programming, arises from the ability of a program to exhibit dynamic behaviour
  – The statements that are executed, and the number of times they are executed can be varied dynamically at runtime, rather than being fixed at compile time

• One of the most important parts of any programming language is the ability to specify that some statements should be repeated a certain number of times

• Java has the three repetition constructs available in the C programming language
  – The for loop, the while loop and the do-while loop
The for loop

- The for loop is the most important looping structure, although it takes some getting used to. It has the following syntax:

```java
for (<initialization>; <boolean-expression>; <post-body update>) {
  <statement-1>
  <statement-2>
  ...
  <statement-n>
}
```

The body of the loop is the collection of statements between the curly brackets.
The initialization part

• The initialization part consists of any Java statement
• It is performed *once only* when execution first reaches the for loop
• It is normally used to initialize a *counter* variable
The boolean-expression part

- The boolean expression controls whether or not the body of the loop is executed.
- The expression is evaluated immediately after the initialization has occurred.
- If its value is true, then all of the statements in the body of the loop are executed in turn.
- If its value is false, then the loop has finished, none of the statements in the body are executed, and execution continues at the first statement after the for loop.
The post-body update

- The post-body update is a Java statement that is executed once *each time* the body of the `for` loop is executed.
- It is executed immediately after the last statement of the body has been executed.
- It is usually used to *update* the counter variable.
The for loop

Before loop

Initialization

Is the boolean expression true?

Yes

Perform Body

Post-body update

No

After loop
The \texttt{for} loop idiom

- \texttt{for}-loops are most often used when we want to do something a specific number of times

```java
for (int i=0; i<5; i=i+1) {
    System.out.println(i);
}
```

\texttt{System.out.println} is a library method that just prints out its argument to the terminal window.
How did this work?

Initialization creates \( i \) and sets it equal to 0
Check if \( i < 5 \), yes
Print out \( i \) – causes 0 to appear on terminal window
Update \( i \) from 0 to 1
Check if \( i < 5 \), yes
Print out \( i \) – causes 1 to appear on terminal window
Update \( i \) from 1 to 2
Check if \( i < 5 \), yes
Print out \( i \) – causes 2 to appear on terminal window
Update \( i \) from 2 to 3
Check if \( i < 5 \), yes
Print out \( i \) – causes 3 to appear on terminal window
Update \( i \) from 3 to 4
Check if \( i < 5 \), yes
Print out \( i \) – causes 4 to appear on terminal window
Update \( i \) from 4 to 5
Check if \( i < 5 \), NO
The increment operator

• The post-body update so often consists of
  \[ i = i + 1; \]
  that there is a short-hand notation for this operation

• The statement \( i = i + 1 \) may be replaced simply by \( i++ \)

```java
for (int i=0; i<5; i++) {
    System.out.println(i);
}
```

• In fact \( i++ \) is both a statement and an expression
  – as an expression it has the value of \( i \) after the incrementing

**Warning:** Either use \( i = i + 1; \) or \( i++ \); but never mix them together and use \( i = i++; \)
A one-statement body

- If the body consists of only one statement, then you can leave out the braces:

```java
for (int i=0; i<10; i++) {
    System.out.println(i*i);
}
```

is the same as

```java
for (int i=0; i<10; i++)
    System.out.println(i*i);
```
Common mistake

• A surplus semicolon! (this mistake is hard to track down)
  ```java
  int i;
  for (i=0; i<10; i++) ;
  {
    System.out.println(i*i);
  }
  ```
  The output from this code is just 100

• Why? What went wrong with the loop?
  – There was a problem with the body
Common mistake

```
int i;
for (i=0; i<10; i++)
{
    System.out.println(i*i);
}
```

The first `for` loop has an **empty body** (just a single semicolon!), while the second shows the “desired” body.

```
int i;
for (i=0; i<10; i++)
{
    System.out.println(i*i);
}
```
Making tables

• Another common use of for loops is to produce tables
• For example, suppose you are asked to produce a temperature conversion table listing the Fahrenheit equivalents of 0°C - 100°C in increments of 5 degrees

<table>
<thead>
<tr>
<th>°C</th>
<th>°F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>15</td>
<td>59</td>
</tr>
<tr>
<td>20</td>
<td>68</td>
</tr>
</tbody>
</table>
A for loop is the solution

```java
int celsius;
int fahrenheit;

for (celsius=0; celsius <= 100; celsius = celsius + 5) {
    fahrenheit = 32 + celsius*9/5;
    System.out.print(celsius);
    System.out.print(" ");
    System.out.println(fahrenheit);
}
```

• Notice
  – the easy-to-read variable names
  – the use of `System.out.print()` to print things without starting a new line
A numerical example

• It is well known that

$$\pi = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots \right)$$

• Suppose that we wish to approximate $\pi$ using this formula

• There are two ways that we can do it
  – approximate by adding up a \textit{fixed number} of terms
  – approximate to with a \textit{certain accuracy}

• The first way is best done using a \texttt{for} loop, and the second way is best done using a \texttt{while} loop
π to a fixed number of terms

```java
public double pi(int n) {
    double approx = 0;
    double mult = 4;
    for (int i=0;i<n;i++) {
        approx = approx + mult/((2*i)+1);
        mult = -mult;
    }
    return approx;
}
```
### Situation at top of loop

<table>
<thead>
<tr>
<th>i</th>
<th>mult</th>
<th>(2*i+1)</th>
<th>approx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.0</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>-4.0</td>
<td>3</td>
<td>4.0</td>
</tr>
<tr>
<td>2</td>
<td>4.0</td>
<td>5</td>
<td>2.666666667</td>
</tr>
<tr>
<td>3</td>
<td>-4.0</td>
<td>7</td>
<td>3.466666667</td>
</tr>
<tr>
<td>4</td>
<td>4.0</td>
<td>9</td>
<td>2.895238095</td>
</tr>
<tr>
<td>5</td>
<td>-4.0</td>
<td>11</td>
<td>3.339682540</td>
</tr>
<tr>
<td>6</td>
<td>4.0</td>
<td>13</td>
<td>2.976046176</td>
</tr>
</tbody>
</table>

Stops when $i$ reaches the requested value
The while loop

- The while loop has the following syntax

```java
while (<boolean-expression>) {
    <statement-1>
    <statement-2>
    ...
    <statement-n>
}
```

- A while loop is usually used when the number of repetitions of the body is not known in advance but depends on some condition being satisfied
The **while** loop

Is the boolean expression true?  

Yes → Perform Body

No →

© CSSE, Gordon Royle 2008
public double pi(double accuracy) {
    double approx = 0;
    double mult = 4;
    double denom = 1;

    while (4/denom > accuracy) {
        approx = approx + mult/denom;
        mult = -mult;
        denom = denom+2;
    }

    return approx;
}
### Situation at top of loop

<table>
<thead>
<tr>
<th>Step</th>
<th>denom</th>
<th>$\frac{4}{\text{denom}}$</th>
<th>approx</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
<td>4.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>3.0</td>
<td>1.3333333333</td>
<td>4.0</td>
</tr>
<tr>
<td>2</td>
<td>5.0</td>
<td>0.8</td>
<td>2.666666667</td>
</tr>
<tr>
<td>3</td>
<td>7.0</td>
<td>0.571428571</td>
<td>3.466666667</td>
</tr>
<tr>
<td>4</td>
<td>9.0</td>
<td>0.44444444444</td>
<td>2.895238095</td>
</tr>
<tr>
<td>5</td>
<td>11.0</td>
<td>0.36363636363</td>
<td>3.339682540</td>
</tr>
<tr>
<td>6</td>
<td>13.0</td>
<td>0.307692307</td>
<td>2.976046176</td>
</tr>
</tbody>
</table>

Stops when the next term is smaller than (or equal to) the desired accuracy
Filled in Squares

- We will use a for-loop to permit our EtchASketch objects to create filled-in squares, rather than just outline squares.
- A square is filled in by drawing a collection of vertical lines, one pixel apart.

```
sc.drawLine(1,1,1,6);
sc.drawLine(2,1,2,6);
sc.drawLine(3,1,3,6);
sc.drawLine(4,1,4,6);
sc.drawLine(5,1,5,6);
sc.drawLine(6,1,6,6);
```
public void fillSquare(int side) {
    for (int i=0; i<side; i=i+1) {
        sc.drawLine(xPos+i,yPos,xPos+i,yPos+side-1);
    }
}

• This is something that could be done with a real Etch A Sketch (given sufficient patience)

• This could also be expressed in terms of the basic methods if desired
An *EtchASketch* picture
The do-while loop

• The final looping construct inherited from C is the do-while loop

```java
do {
    <statement-1>
    <statement-2>
    ...
    <statement-n>
} while (<boolean-expression>);
```
The do-while loop

Is the boolean expression true?

Yes

Perform Body

No

Perform Body
CITS 1200 Java Programming

Selection
Scope of this lecture

• The if statement
• A logic game
• The switch statement
• Random walks
Dynamic Behaviour

- Recall that the ability of a program to exhibit dynamic behaviour is one of the main reasons underlying the power of computer programs.
- Dynamic behaviour arises from the use of control structures which alter the default sequential execution of statements.
- The for, while and do-while loops permit the repetition of blocks of code.
- The if and switch statements permit the dynamic selection of blocks of code.
- Selection statements allow programs to exhibit logic!
The if statement

• The most common form of the if statement is

    if (<boolean-expression>) {

        <statement-block-1>

    } else {

        <statement-block-2>

    }
**Behaviour of if**

- When the `if` statement is encountered, the boolean expression is evaluated.
- If it is `true`, then `<statement-block-1>` is **executed**, and `<statement-block-2>` is **skipped**.
- If it is `false`, then `<statement-block-1>` is **skipped**, and `<statement-block-2>` is **executed**.
- A boolean expression **must** be either `true` or `false`, so **exactly one** of the two statement-blocks is executed.
- Execution then proceeds directly to the next statement after the `if` statement (there is no repetition involved).
Omitting the `else`

- The `if` statement is used to choose between two alternative courses of action.
- Often though, one of the alternatives is simply to do nothing – in this situation, the `else` can be omitted.

```java
if (numDeposits + numWithdrawals > 10) {
    int fee = numDeposits + numWithdrawals - 10;
    this.withdraw(fee);
}
```

- In this example, the fee is charged only if there are excess transactions – if the condition is `false`, then nothing needs to be done.
Guess the number

• Consider the child’s game “Guess the Number”
• One person secretly chooses a number between 1 and 100
• The second person guesses the number, and is then told either
  – “Too low”
  – “Correct”
  – “Too high”
  depending on whether their guess was too low, correct or too high
**GuessTheNumber**

- We will design a class to play the game, with the computer choosing the number and the human user being the guesser.
- Each object of this class will represent one game.
- Therefore each object will have to choose a number randomly, store it in an instance variable, and respond correctly to the user’s guesses.
- As always, we must consider
  - the instance variables
  - the constructors
  - the methods
Instance Variables

• We only need one instance variable to store the secret number

    public class GuessTheNumber {

        private int secretNumber;

    }

    }
Constructor

- The role of the constructor is to set up the object so that it is “ready” to perform its actions.
- Therefore, the constructor should choose the secret number randomly.
- To do this, we will use a library class called `java.util.Random`.
- Objects of this class generate *pseudo-random* numbers:
  - True random numbers are hard to find, so computers use mathematical formulas to generate a sequence of numbers that have similar statistical properties to genuine random numbers.
Using `java.util.Random`

The constructor is simple enough

and there's the method we need
Code for the constructor

```java
import java.util.Random;

public class GuessTheNumber {
    private int secretNumber;

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

- Allows us to use the “short name” Random
- Declares and creates a new random number generator `r`
- Asks `r` for a random int between 0 and 99, and then adds 1 to get a number between 1 and 100
The import statement

• The import statement is a convenient short-cut to save you from typing the full name of every class you use.
• Every class in Java has a full name based on which package it belongs to:
  – java.util.Random
  – java.awt.Color
  – java.lang.String
• You can always use the full name of the class, for example when declaring or initializing variables:

```java
java.util.Random rg;
rg = new java.util.Random();
```
The short name

• An import statement gives the compiler a clue about how to translate a short name to a full name - if the compiler encounters a “short name” like Random that is not the name of a class in the current project,

    Random r = new Random();

it will then check the import statements.

• If the class file includes at the top

    import java.util.Random;

then it will assume that every occurrence of Random refers to an object of the class java.util.Random
More import

• If you use a lot of classes from one particular package, you can just put
  
  import java.util.*;

• Then the compiler will check every class in that package to find a matching name

• The package java.lang contains very fundamental classes and these can always be referred to by their short name
  – String is always taken to mean java.lang.String
  – Math is always taken to mean java.lang.Math

• If your project uses two classes with the same short name, then you cannot avoid using full names
Method

• The class only needs one method – one to check the user’s guess and respond whether or not it is too high or too low or just right

• There are several ways in which the user can be informed of how their guess worked out – the method could return a value, or it could display the result on the terminal window or even graphically

• We will start simply with

```java
public String checkMyGuess(int guess)
```

• This will return a `String` – either “Too High”, “Too Low” or “Well Done”
public String checkMyGuess(int guess) {
    String response;
    response = "Well Done";

    if (guess > secretNumber) {
        response = "Too High";
    }

    if (guess < secretNumber) {
        response = "Too Low";
    }

    return response;
}
public String checkMyGuess(int guess) {
    if (guess == secretNumber) {
        return "Well Done";
    } else if (guess > secretNumber) {
        return "Too High";
    } else {
        return "Too Low";
    }
}
Java can be over-careful!

```java
public String checkMyGuess(int guess) {
    if (guess == secretNumber) {
        return "Well Done";
    }
    if (guess > secretNumber) {
        return "Too High";
    }
    if (guess < secretNumber) {
        return "Too Low";
    }
}
```

The compiler cannot be certain that one of the return statements will be executed, so it fails to compile.
Counting Guesses

- Most versions of “Guess The Number” count the number of guesses that the guesser needs.
- As the number of guesses must be “remembered” between method calls, it must be stored in an instance variable.

```java
private int secretNumber;
private int numGuesses;

public GuessTheNumber() {
    Random r = new Random();
    secretNumber = r.nextInt(100)+1;
    numGuesses = 0;  // Extra instance variable
}
```

Initialize to 0 in constructor
Updating and reporting

• At the beginning of the `checkMyGuess` method, the counter should be incremented
  
  ```java
  numGuesses = numGuesses + 1;
  ```

• The idea is that each time the user calls `checkMyGuess`, they have made another guess, so the counter is increased

• Finally, when the game is over, the user should be informed of how many guesses they took:
  
  ```java
  if (guess == secretNumber) {
      response = "You took " + numGuesses + " guesses";
  }
  ```

The `+` operator is used to **concatenate Strings** – in this case it combines three Strings into a sentence, which is then returned to the user.
What is a good score?

How many guesses do you need to guarantee that you can get the number?
The `switch` statement

- The `switch` statement is a complex selection statement (inherited from the C programming language)
- It is used when there are more than two or three alternatives to choose from, to avoid an extended sequence of `else-if-else-if-else-if` statements
Syntax of switch

switch (<int-expression>) {
  case literal-1:
    <case-1-statements>
    break;
  case literal-2:
    <case-2-statements>
    break;
    ...
  case literal-n:
    <case-n-statements>
    break;
  default:
    <default-statements>
    break;
}

© CSSE, Gordon Royle 2008
What happens in a switch?

- As soon as the switch statement is reached, the expression of type int is evaluated.
- If the value matches any of the cases that are tagged with literals, then execution starts at that point, and continues until a break statement is reached.
- If the value does not match any of the literals, then the statements under the default case are executed.
- As soon as a break statement is reached, the switch statement is finished.
- If a break statement is omitted, execution will simply continue over multiple cases.
Random Walks

• A random walk is a path taken by a particle that is equally likely to move in any direction
  – Sometimes, it is more colourfully described as the path taken by a drunk who randomly staggers in any direction, independent of the direction of his last step
• Random walks are very important in computer simulation of natural phenomena, such as Brownian motion, percolation, diffusion and movement of share prices
• We will consider a simple discrete simulation of a random walk, with a single particle

A list of books about random walks:
http://www.ericweisstein.com/encyclopedias/books/RandomWalks.html
Discrete simulation

• In a discrete simulation, time passes in discrete steps and at each step the position of the particle is updated
• Although there are many options, we consider the simplest one first
  – The particle occupies a position with integer co-ordinates on a 2d-grid
  – At each time-step, the particle takes a single step of fixed length either up, down, left or right independently of its position or of any previous moves
• We will use the EtchASketch to visualize the process of the random walk!
How many steps?

• An obvious question is how many steps to simulate
• One solution is to assume that the world is bounded and has absorbing walls – the particle is absorbed if it reaches the edge and the walk finishes
• This is an ideal situation for a while loop – the idea is to have a condition that expresses the property that the particle is within the bounds of the world, and to continue only while that condition remains true
• This condition requires that
  
  \[ 0 \leq xPos < width \]
  \[ 0 \leq yPos < height \]
public void randomWalk(int stepSize) {
    java.util.Random r = new java.util.Random();

    while (0 <= xPos && xPos < 400 && 0 <= yPos && yPos < 400) {

        switch(r.nextInt(4)) {
            case 0: up(stepSize); break;
            case 1: left(stepSize); break;
            case 2: right(stepSize); break;
            case 3: down(stepSize); break;
        }
    }
}
CITS 1200 Java Programming

Arrays
Scope of this lecture

• Arrays
• Constructing arrays
• Basic array uses
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]` and so on; each element is a separate variable
Use of arrays

- Arrays are used when we have large numbers of identical objects that we want to operate on as a collection
  - A collection of student marks that we want to analyse
  - A collection of temperatures that we want to average
  - A collection of names that we want to sort

Warning: In Java, the indexing of an array starts at 0, so the first element of the array `a` is `a[0]`
Declaring arrays

• An array is declared using similar syntax for other variables:

```java
int[] a;

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

```java
double[] temps;

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

```java
String[] names;

• Declares names to be a variable representing an array of Strings
```
Simple example

- The Bureau of Meteorology (www.bom.gov.au) collects weather data recorded at many different locations e.g.

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

- This data consists only of the average monthly maximum temperatures, average monthly minimum temperatures and average monthly rainfall, although there is much more available.
How should this be stored?

• Data of this form is very naturally modelled in OOP – we will design a `WeatherStation` class, where each object represents a single weather station

• What should the attributes be?

```java
public class WeatherStation {
    private String name;

    private double janMax;
    private double febMax;
    private double marMax;
    // many omitted variables
    private double decRain;
}
```

It would be a nightmare to store and manipulate 36 different variables!
And imagine if the WeatherStations collected *daily* data!!
Store the data in arrays

public class WeatherStation {

    private String name;
    private double[] maxTemp;
    private double[] minTemp;
    private int[] rain;

    // constructors & methods omitted

}
Creating Arrays

• An array is an object in a Java program
• Therefore the declaration simply creates a reference to refer to the array, but does not create the array itself
• Hence, the declaration
  ```java
  int[] a;
  ```
  creates a shoebox called `a`, big enough to hold an object reference, and currently set to the special value `null`

Only space for the array reference has been created, not the array itself
Creating arrays II

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

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

An object containing 7 variables of type `int` is created
Creating arrays III

• The seven variables do not have individual names, but are referred to by the array name, and an index
Referencing array elements

- Array elements can be used in just the same way as any other variable of that type

```java
a[4] = 15;
a[2] = 7;
a[6] = a[0] + 17;
```
Indexing array elements

- The power of arrays comes from the fact that the index can be a variable or expression

```java
int x = 3;
a[x] = 5;
a[7-x] = 44;
```
WeatherStation constructor

- The constructor for the WeatherStation requires the client to specify the values for that particular location

```java
WeatherStation(String name, double[] maxTemp, double[] minTemp, int[] rain) {
    this.name = name;
    this.maxTemp = maxTemp;
    this.minTemp = minTemp;
    this.rain = rain;
}
```

The client must specify these values when constructing a new object of the class WeatherStation
In BlueJ

- BlueJ allows you to use an *array literal* to specify an array – this is just a comma-separated list of the elements in curly brackets.
- In general, for reasons that are unclear, array literals are only allowed in Java for initialization.
Remember the Golden Rule

- An array behaves like an object, and so the name of an array is actually a *reference*.
- Therefore a statement of the form
  ```java
  this.maxTemp = maxTemp;
  ```
  copies a *reference* into the instance variable `maxTemp`.
- One consequence of this is that both the client and the `WeatherStation` object have a reference to a *single copy* of the actual data.
- This is not a problem for data that will be used without being altered.
WeatherStation methods

- The methods of the WeatherStation will provide clients with a variety of information about that location.
- For example, clients may wish to know:
  - The annual average rainfall
  - The annual average of the daily maximum temperature
  - The dryest month, hottest month, coldest month etc.
Total rainfall

• To find the average annual rainfall for a location, we simply add up the rainfall for each of the twelve months

```java
public int annualRain() {
    return
}
```

This works, but is unsophisticated, inelegant, tedious to type and error-prone
Use a for loop

• Arrays and for loops go together like toast and vegemite!

```java
public int annualRain() {
    int total = 0;
    for (int i=0; i<12; i++) {
        total = total + rain[i];
    }
    return total;
}
```

The body of the for loop is executed 12 times, each time adding a different array element to the total.
Averages

The average maximum temperature over the whole year is obtained by adding the 12 values and dividing by 12
– there is a slight inaccuracy due to the different length of each month, but we will ignore that for now

```java
public double averageMaxTemp() {
    double total = 0;
    for (int i=0; i<12; i++) {
        total = total + maxTemp[i];
    }
    return total / 12;
}
```
More averages

```java
public double averageMinTemp() {
    double total = 0;
    for (int i=0; i<12; i++) {
        total = total + minTemp[i];
    }
    return total / 12;
}
```

This works, but it somehow seems silly writing two methods that do almost the same thing
Finding the length of an array

- When an array is created, its length is determined (unlike languages such as C), and we can find it out

- The expression `a.length` is the number of elements in the array `a`
  - We can imagine that the object `a` has a variable called `length` and therefore `a.length` gives us the value of that variable
  - Do not get confused into thinking that this is a method – `a.length()` will not work!
The array idiom

- This construct recurs so often in dealing with arrays that it is worth explicitly describing:

```java
for (int i=0; i<a.length; i++) {
    // process element a[i]
}
```
The general average method

• Instead, we write one method that calculates the average of any array of any length

```java
private double average(double[] a) {
    double total = 0;
    for (int i=0; i<a.length; i++) {
        total = total + a[i];
    }
    return total / a.length;
}
```

This is called a helper method – they are usually made private as they are for the “internal use” of the class itself.
Using the helper

- Both of the “average temperature” methods can now use the same private helper method, simply by calling it with the appropriate array as an argument – this means that the code to calculate averages is located in only one place.

```java
public double averageMaxTemp() {
    return this.average(maxTemp);
}

public double averageMinTemp() {
    return this.average(minTemp);
}
```
**Duplicate code**

- In general repeated, or similar, code should always be replaced by a private helper method – this process is called *factoring out* common code
- This has a number of advantages
  - testing and debugging need only be performed once
  - subsequent changes to implementation need only be performed in one place
  - duplicating code via cut-n-paste is a major known source of hard-to-find bugs
private double max(double[] a) {
    double max;
    max = a[0];
    for (int i=1; i<a.length; i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}
Public methods use the helper

```java
public double highMaxTemp() {
    return this.max(maxTemp);
}

public double highMinTemp() {
    return this.max(minTemp);
}
```
What about the rainfall?

- Unfortunately, as the rainfall data is held in an `int[ ]` we cannot use the existing private helper method, but need another one to deal with `int`’s

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

This is a fundamental problem/issue introduced by having the primitive types represented by value; there are some “pure” object oriented languages where `everything` is an object including numbers, but this makes them run too slowly for some practical problems.
CITS 1200 Java Programming

Arrays II
Scope of this lecture

• Using arrays
• Array subtleties
• Arrays as parameters and return types
• Arrays of objects
Using arrays

- A primary use of arrays is to store sequences of data for analysis and/or display
- Examples:
  - Student marks (int[])
  - Monthly maximum temperatures (float[] or double[])
  - Daily closing prices of a share (float[] or double[])
  - Weekly sales figures for a shop (float[] or double[])
Returning arrays

• An array is just another object in a Java program, and so a method can return an array
• The following method returns an array containing the first $n$ cubes

```java
public int[] cubes(int n) {
    int[] c = new int[n];
    for (int i=0; i<n; i++)
        c[i] = i*i*i;
    return c;
}
```
public int[] cubes(int n) {
    int[] c = new int[n];
    for (int i=0; i<n; i++) {
        c[i] = i*i*i;
    }
    return c;
}
In BlueJ

Object result

Int length

0
1
8
27
64
125
216

result.result : int[]

Inspect
Get

Show static fields

Close
References and arrays

What will the result of the following code be?

```java
int[] a = new int[5];
int[] b = new int[5];

for (int i=0; i<5; i++)
    a[i] = i*i;

for (int i=0; i<5; i++)
    b[i] = i*i;

if (a == b)
    System.out.println(“Equal”)
else
    System.out.println(“Not Equal”);
```

Are these two arrays equal or not?
Why are they different?

• The == operator simply works on the references! And because there are two objects, the references are different.
Copying arrays

• How do we make a copy of an array?

```java
int[] a = new int[5];
int[] b = new int[5];
for (int i=0;i<5;i++)
    a[i] = i*i;

b = a;
```

This looks like a promising way to copy one array to another - but does it work?
Work through step by step

- After the `for` loop, this is the situation
Assignment just assigns the reference!

- The statement `b = a` just copies the value of the reference to `b`

This object is now an “orphan” and will be marked as “ready for garbage collection”
You must copy element by element

- To truly copy the arrays, you must copy each element
  
  ```java
  for (int i=0; i<5; i++)
      b[i] = a[i];
  ```
2-dimensional arrays

- We can declare and use arrays of more than one dimension!

```java
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

© CSSE, Gordon Royle 2008
A 2-d array is really an “array of arrays” - an array of length 4, where each element is an array of length 3.
2-dimensional arrays

Of course the computer does not actually store the entries in “rows” and so we should just view this 2d array as being 12 variables arranged in four groups of three.
Here is some simple code just to show 2-d arrays in BlueJ

```java
public int[][] timesTable(int n) {
    int[][] times = new int[n][n];
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            times[i][j] = i*j;
        }
    }
    return times;
}
```
**BlueJ: Method Result**

- **Object result**: 
  - **Inspect**
  - **Get**
  - **Close**

**BlueJ: Object Inspector**

- **result.result : int[][]**
  - **int length**: 6
  - [0]
  - [1]
  - [2]
  - [3]
  - [4]
  - [5]

- **Inspect**
- **Get**
- **Show static fields**
- **Close**

© CSSE, Gordon Royle 2008
Arrays of objects

• An array can have any type as its base type, either primitive type or reference type
• When using an array 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 of reference 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 in a loop
Arrays of Students

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[] unitList;

unitList = new Student[numStudents];

unitList[0] = new Student("042371X", 64);
unitList[1] = new Student("0499731", 72);
unitList[2] = new Student("0400127", 55);
...
unitList[numStudents-1] = new Student("0401332", 85);
The three steps
Using arrays of objects

- Using arrays of objects just requires remembering that each element of the array is (a reference to) an object

```java
public Student topStudent(Student[] unitList) {
    Student top;
    top = unitList[0];
    for (int i=1; i<unitList.length; i++) {
        if (unitList[i].getMark() > top.getMark()) {
            top = unitList[i];
        }
    }
    return top;
}
```
private int max(int[] a) {
    int max;
    max = a[0];
    for (int i=1; i<a.length; i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}
Method signatures

- Method signatures
  - int max(int[] a)
  - and
    - Student topStudent(Student[] unitList)
Initialization

```java
int max;
max = a[0];
```

Declare a variable to hold the “best so far” and initialize it to the first element in the array

```java
Student top;
top = unitList[0];
```
Initialization

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

Check each element in turn, compare it with the best so far, and update the best so far if necessary

for (int i=1; i<unitList.length; i++) {
    if (unitList[i].getMark() > top.getMark()) {
        top = unitList[i];
    }
}
Return

```
return max;
```

Finally *return* the extreme element – the highest *int* or the *Student* with the best mark

```
return top;
```
Highest mark

• What if the user wishes to find the highest mark?
• Two options
  – Return the Student object, and let the user call getMark to find it out
  – Provide another simple method that uses topStudent as a helper

```java
public int highestMark(Student[] unitList) {
    return this.topStudent(unitList).getMark();
}
```
Common array issues

• The commonest problem with an array is using an array index that is out of bounds, either less than 0 or larger than the largest index.

• In particular, if we have an array

```java
int[] a = new int[6];
```

then any attempt to access `a[6]` will generate an `ArrayIndexOutOfBoundsException` and execution will be terminated.

• Another common problem is declaring and creating an array of objects, but forgetting to populate it

  – any attempt at making a method call such as `unitList[i].getMark()` will then result in a `NullPointerException`
CITS 1200 Java Programming

Bugs and Debugging
Scope of this lecture

• Bugs
• Syntax Errors
• Runtime Errors
• The BlueJ debugger
Errors

• Programming is an error-prone process
• We identify three types of error
  – A syntax error occurs when the program cannot be compiled
  – A runtime error occurs when the program compiles, but crashes with an error message from the runtime environment
  – A logic error occurs when the program compiles and runs, but produces incorrect answers
• An error in a computer program is universally known as a bug and eliminating errors from the program is known as debugging
Syntax Errors

• Syntax errors are the most common, but easiest to eliminate because they are detected by the compiler

• When the code cannot be compiled, the compiler will halt with an error message
  – The compiler tries to give its best explanation of why it cannot continue with the compilation process
  – Sometimes this explanation is difficult to interpret, but learning to use these error messages effectively will save a vast amount of time
  – A substantial number of syntax errors are caused simply by errors in typing or layout
cannot resolve symbol

The compiler does not understand what some name means - here the name xpos is causing the problem
Extra help from BlueJ

The most common problem is simply mistyping the name of the variable - in this case `xpos` instead of `xPos` - other common problems include forgetting to declare the variable before using it, getting the name of a class wrong or using a class that is not in the current project.
class or interface expected

Accidentally putting a new method *outside* the class, rather than between the opening and closing curly brackets - Java thinks that this must be the start of a new class, and so it is expecting the word `class`
Type checking

The compiler will work hard to make sure that all of the variables are used appropriately for their type.
Return Types

- The compiler will check that anything returned from a method has the right type.

```java
//
public class BankAccount {
    private int balance;
    private int accountNumber;
    private String accountName;

    public BankAccount(String accName, int accNum, int initBalance) {
        balance = initBalance;
        accountNumber = accNum;
        accountName = accName;
    }

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

Method claims to return an int, so the return statement cannot attempt to return a String!
Other syntax errors

• Some other common syntax errors are caused by omitting common symbols - usually you will have to look at the previous few lines to find the mistake
  - } expected
  - { expected
  - ; expected

• There are a number of other less common syntax errors, but eliminating syntax errors is usually fairly straightforward because of the information given by the compiler
Runtime Errors

• Runtime errors are problems that occur while the program is running - these are much harder to track down than syntax errors, particularly if they occur after the program has been running for some time

• The main tools to help isolate and correct runtime errors are the error messages and the interactive BlueJ debugger

• Probably the most common error is the infamous

    NullPointerException
Suddenly a method call causes a crash with the message `NullPointerException` - the offending line is highlighted by BlueJ.
What has happened?

• The line that caused the problem was

```java
sc.drawLine(xPos,yPos,xPos,yPos-distance);
```

• This is an instruction to the object `sc` to draw a line
• A `NullPointerException` means that the variable `sc` contains the value `null` rather than a valid reference to an object
• What could have caused this?
Variable shadowing

• By far the most common cause of a NullPointerException is accidental variable shadowing

```java
private SimpleCanvas sc;
private int xPos;
private int yPos;

public EtcSketch() {
    SimpleCanvas sc = new SimpleCanvas();
    xPos = 200;
    yPos = 200;
}
```

• The constructor declares a new local variable `sc` that is different to the instance variable `sc`
On construction

• When the EtchASketch object is constructed
  – The computer creates an instance variable sc and sets it equal to the default value null, and then calls the constructor
  – The constructor should just create the SimpleCanvas and then assign it to the existing instance variable sc
    
    sc = new SimpleCanvas()
  – Instead, the constructor executes a combined “declare and initialize statement” that has the same effect as
    
    SimpleCanvas sc;
    sc = new SimpleCanvas();
  – This creates a new local variable sc that “shadows” the instance variable sc so the instance variable never gets changed from null
Deliberate Shadowing

• Suppose that we want to add a constructor to `EtchASketch` to allow the user to specify its size

```java
private SimpleCanvas sc;
private int xPos;
private int yPos;
private int width;
private int height;

public EtchASketch(int inWidth, int inHeight) {
    width = inWidth;
    height = inHeight;
    sc = new SimpleCanvas("Etcher",width,height);
    xPos = width/2;
    yPos = height/2;
}
```
“Artificial” variable names

```java
public EtchASketch(int inWidth, int inHeight) {
    width = inWidth;
    height = inHeight;
    sc = new SimpleCanvas("Etcher",width,height);
    xPos = width/2;
    yPos = height/2;
}
```

• The two variables `inWidth` and `inHeight` are just used to transfer the information to the instance variables

• It is tedious and error-prone to have to come up with slightly different names just for the one line of code where they are each used

© CSSE, Gordon Royle 2008
Solution: use the same names!

private SimpleCanvas sc;
private int xPos;
private int yPos;
private int width;
private int height;

public EtchASketch(int width, int height) {
    this.width = width;
    this.height = height;
    sc = new SimpleCanvas("Etcher", width, height);
    xPos = width/2;
    yPos = height/2;
}
Another use of `this`

- “Assign the value of the `local` variable `width` to the `instance` variable `width` belonging to `this` object”

```java
this.width = width;
```

- In general, a variable declared `inside` a method shadows an instance variable of the same name, so you must use `this` to `explicitly` refer to the instance variable if necessary
The debugger

• Sometimes an error is not so easy to track down and occurs somewhere in the middle of a complicated piece of code
• BlueJ has a debugger that allows you to stop execution at any line of code, examine the values of the variables at that point, and execute one line of code at a time to identify the precise point at which things go wrong
Breakpoints

- Click in the left column after the class has been compiled at the statement where you want execution to stop
- In this case, we have set a breakpoint in the constructor
Execution halts

```java
public class EtchASketch {
    private SimpleCanvas sc;
    private int xPos;
    private int yPos;
    private int width;
    private int height;

    public EtchASketch(int width, int height) {
        this.width = width;
        this.height = height;
        SimpleCanvas sc = new SimpleCanvas("Etcher", width, height);
        xPos = width/2;
        yPos = height/2;
    }

    public EtchASketch() {
        SimpleCanvas sc = new SimpleCanvas();
        xPos = 200;
        yPos = 200;
    }

    public void up(int distance) {
        sc.drawLine(xPos, yPos, xPos, yPos-distance);
    }
}
```
And all the information is shown

- The *instance variables* all have their default values (null, 0)
- The *local variables* have the values specified by the user
Step-through

- After hitting Step a few times, it becomes clear that a new `sc` variable has been created and the instance variable `sc` has not been altered from its default `null` value
Logic Errors

• The final class of errors is the hardest to detect and correct
• The program compiles and runs and either
  – Produces obviously incorrect output, or worse
  – Produces seemingly correct output
• These sort of errors can only be detected with thorough testing of the resulting program
  – A programming practice called eXtreme Programming emphasizes a continuous cycle of “code-a-little, test-a-little” throughout development of code
• Later we will see a few language constructs that help in this process
Numerical errors

- Java has several *integer* numeric types and 2 *floating point* numeric types
- Integers can be represented exactly in the computer, and so integer arithmetic is perfectly accurate

\[24755 + 181032 = 205787\]

\[
\begin{array}{c}
00000000000000000110000010110011 \\
+ \\
00000000000000000101100001100101000 \\
= \\
0000000000000000011001000111011011
\end{array}
\]
Overflow

• Although integer arithmetic is exact, it is still the case that only a finite number of values can be represented.

• In particular, the largest possible value of an `int` is the number `2147483647`.

• So what happens if we deliberately create an “overflow”

```java
int x;
x = 2147483647;
x = x + 1;
```

• What actually happens is that the calculation silently “rolls over” and the resulting value is the large negative value `−2147483648`.

Should this cause a runtime error?
Integer Division

• Another very frequent source of errors in Java programs is forgetting how integer division works
• If the two arguments are integers, then the result is an integer obtained by *truncating* the true value
• For example, suppose that
  – `int numPassed;`
  – `int numStudents;`
  are variables representing the number of students who passed a unit and the number of students taking the unit
• Which expression correctly gives the *percentage* pass rate (as in 77.9% of students passed the unit)?
Integer Division cont.

1. return 100 * numPassed / numStudents;
2. return numPassed / numStudents * 100.0;

1. return (double) numPassed * 100 / numStudents;
2. return 100d * numPassed / numStudents;
3. return 100* (double) numStudents / (double) numPassed;
Floating point arithmetic

- Real numbers, such as $\frac{1}{3}$ or $\sqrt{2}$ or $\pi$ have infinitely long decimal expansions, but the computer can only store a fixed finite number of places.
- For example, $\pi$ is available as `Math.PI` and has the value $3.141592653589793d$.
- Therefore any calculation involving a real number is not perfectly accurate, but has a small round-off error.
- In some numerical applications, a small initial error can accumulate and end up dominating the entire calculation.
What is 0.2+0.1?

The value is incorrect in the last decimal place. Any computations with floating point numbers must be treated very carefully because of potential numerical inaccuracy.
CITS 1200 Java Programming

Class Variables and Methods
Scope of this lecture

• Class methods
• static
• Examples from the Java API
• Class variables
Target objects can be vital...

- Consider a method like `getBalance` in `BankAccount`
  ```java
  public int getBalance() {
      return balance;
  }
  ```
- When this method is called, a target object must be specified, because we are finding the balance of a specific `BankAccount`
  ```java
  int value = savings.getBalance();
  ```
- The result of the method call will almost certainly be different if a different target object is specified
... or irrelevant

- On the other hand, consider a method like `isPrime`

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

- The result of this method call will depend only on the client specified value of `n`, regardless of which target object the method call is made on
What is the difference?

• The method `getBalance` uses an *instance variable*, while the method `isPrime` doesn’t.
• The method `getBalance` is enquiring about a *particular* object, while `isPrime` is asking for a calculation.
• “Peter, what is your student number?”
  – Refers to information about Peter himself.
  – Answer will be different if “Peter” is replaced with “Jill”.
• “Peter, what is the date of the midsemester test?”
  – Does *not* refer to any information about Peter.
  – Answer will be the same if “Peter” is replaced by any other object.
All methods are in classes

• All methods must belong to some class, so the `isPrime` method still has to be defined in a class

```java
public class PrimeFinder {

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

• The class has no instance variables, and no constructor
• It is really just a “placeholder” for the method
Using the method

• In order to use the `isPrime` method, the client still has to create an object of the class `PrimeFinder` just in order to have a target object for the method call

```java
PrimeFinder pf = new PrimeFinder();
if (pf.isPrime(97)) {
    // do something
}
```

• The object `pf` plays no particular role at all
static

- In this situation, if a method involves no instance variables, the programmer can choose to declare it to be static, and it can then be called without creating any objects of that class.

```java
public class PrimeFinder {
    public static boolean isPrime(int n) {
        for (int i=2; i<n; i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```
Calling a static method

• If a method has been declared to be static, then the client can use it by specifying the class name as the target

    TargetClass.methodName();

• For the isPrime example, a method call could now look something like

    if (PrimeFinder.isPrime(97)) {
    // do something
    }

© CSSE, Gordon Royle 2008
In BlueJ

- In Blue, you right-click on the **class** in order to run a static method, rather than on an object.

- This is to emphasize that a static method “belongs” to the class and not individual objects of that class.
Instance and class methods

- A method that involves an instance variable cannot be declared to be static, because it must have a target object
  - we normally think of each object as having its “own copy” of the method, and being responsible for running it
- Therefore, non-static methods are also called instance methods or object methods – they “belong” to individual instances
- On the other hand, a static method can be called even if there aren’t any objects of its class currently in existence, so we think of the method as “belonging” to the entire class
- Therefore static methods are also called class methods
Use of class methods

- Essentially class methods are used as the equivalent of a “pure function” – something that just requires a calculation involving only the client-supplied data.
- The library class `Math` contains *only* static methods, because they are all just pure functions.

The return type is listed as `static double` or `static int` etc. to indicate that it is a class method.
Using the class Math

- Whenever you need a mathematical function, it will (probably) be the in the class Math
- For example, Java does not have a power operator, but it is available in Math

```java
class AreaCalculator {
    public static double circleArea(double radius) {
        double area = 3.14159 * Math.pow(radius, 2);
        return area;
    }
}
```

This is poor quality code!
Utility Classes

• A class like `Math` that contains only static methods is sometimes called a utility class, because it just provides “utility” methods for use in other classes

• There is no point in ever creating an object of the class `Math` because it can never do anything that the existing methods cannot do

• In fact it has been made impossible to create an object of the class `Math`!
  – this is done by giving a dummy constructor, but making it `private`
Other class methods in the API

- There are a number of useful class methods in the API
- The class `String` has a collection of related class methods

```java
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(float f)
static String valueOf(double d)
static String valueOf(int i)
static String valueOf(long l)
```
What are these used for?

- These methods are used to obtain the String representation of primitive values, often for graphical display.
- For example, consider the method from `SimpleCanvas`:
  
  ```java
drawString(String text, int x, int y)
```
- Suppose the result of some calculation is stored in the variable `result` of type `double`, and the programmer wants to display it on a `SimpleCanvas`.
- This can easily be done by using the `valueOf` method:
  
  ```java
c.drawString(String.valueOf(result), x, y);
```
  
  where `c` is a variable of type `SimpleCanvas`, and `x` and `y` are `int` variables containing the position co-ordinates.
Non-BlueJ programs

• In a program that is run in BlueJ, the user is a participant in the running program and so can interactively call methods, specify parameters and receive results.

• However BlueJ is just a development environment and the code that you write is pure Java that can be run separately from BlueJ.

• The only question is how to start a program outside of BlueJ?

• The answer is that you must give at least one of your classes a special class method called `main`.
The main method

• This class method must have the following signature
  \[\text{public static void main(String[]} \text{args})\]
• This method is the starting point for all Java programs that are not run in an interactive IDE like BlueJ – the user specifies a class and the runtime environment starts execution at the main method in this class
• Just using Java with no IDE, the user types (at the Windows or Linux command prompt)
  \[\text{java ClassName}\]
  and execution starts at the main method of that class
Running non-BlueJ programs

• In a program run outside BlueJ, the user cannot easily interact with the program, but just sets it running
• If parameters need to be specified at any point, then the user must specify *all* the parameters at the beginning of the program
• If the program needs to be interactive, then the program must *itself* create user-interface components (buttons, scrollbars, textfields etc) in order to communicate with the user while the program is running
Class Variables

- As well as having methods associated with an entire class rather than with the individual objects of the class, it is also possible to have variables that are shared across an entire class.
- As an example, let's consider adding interest to a BankAccount.
- The basic code is easy enough:
  ```java
  public void addInterest() {
    int interest = balance * interestRate/100;
    balance = balance + interest;
  }
  ```
- But where and how should we declare the interest rate?
The options

- There are three main options:
  - Have an instance variable called `interestRate`
    - Each instance has a copy of this variable
    - But they are all the same! The Bank has only one interest rate
  - Have a parameter called `interestRate`
    - This is a reasonable option
      ```java
      public void addInterest(int interestRate)
      {
        // code as before
      }
      ```
  - The third option is to have a class variable called `interestRate`
Class variables

- A class variable is a variable that belongs to an entire class of objects
- There is only one copy of the variable, and every object of the class shares it
Where “is” the class variable?

- Think of the class `BankAccount` as a factory that makes `BankAccount` objects according to the class description.

- The class variable is a shoebox that is stored “at the factory” separately from the objects.
How do we declare a class variable

- A class variable is declared in the same place as an instance variable, and you indicate to the compiler that it is a class variable by using the word `static`

```java
public class BankAccount {
    private int balance;
    public static int interestRate;

    // the rest of the code
}
```
How do you refer to a class variable

• There are three ways of referring to this variable
• Each object of the class `BankAccount` can just refer to `interestRate`
  – From inside an object it just looks like an instance variable – except that it is *shared* – if one object changes it, then it is changed for everyone!
• If it is public, then any other object can refer to it as `BankAccount.interestRate`
  – i.e “the variable `interestRate` belonging to the class `BankAccount`”
• If `b` is *any* object of class `BankAccount`, we can use `b.interestRate`
Constants

• Class variables are often used to provide access to constants – values that are frequently used but not changed
• Constants can be numerical values
  – Math.PI
  – Math.E

```java
public static double circleArea(double radius) {
    double area = Math.PI * Math.pow(radius, 2);
    return area;
}
```
Constant Objects

- Constants can also be frequently used objects
- The class `java.awt.Color` makes available a number of “pre-constructed” objects
  - `java.awt.Color.RED`
  - `java.awt.Color.BLUE`
  - etc.
  - `java.awt.Color.BLACK`
- You can use these colours without the expense of constructing them from scratch
Magic Numbers

- A *magic number* is a programmer-chosen literal that appears in the source code.
- For example, a programmer might decide that a `BarChart` object should use a `SimpleCanvas` of size 400 x 400.
- This choice may result in many occurrences of 400, or related literal values in the code – the *magic* numbers.
  - The constructor may have `new SimpleCanvas(400, 400);`
  - The scaling method might always scale the values according to 400 pixels being the maximum height.
  - The `clearCanvas` method may rely on the size being 400 x 400.
**But what if something changes..**

- But now what happens if the boss comes in and says that 400 is too small, and that it should now be 500 pixels wide and high.
- Someone must search through the code and replace every 400 by 500 – this cannot be done automatically because there may be *other* literals equal to 400, but for an unrelated part of the program.
- Worse still, the programmer may have used 399 for the right-most pixel, or 200 for half-way across the `SimpleCanvas` etc.
Symbolic Constants

• The solution is to use *symbolic constants* wherever possible

```java
private static int CANVAS_SIZE = 400;
```

• Then everything that refers to the size of the `SimpleCanvas` should be expressed symbolically
  – `CANVAS_SIZE` for 400
  – `CANVAS_SIZE - 1` for 399
  – `CANVAS_SIZE / 2` for 200 etc.

• If change is needed, then it can be done by replacing a single line of code, and everything else scales automatically!
CITS 1200 Java Programming

Characters and Strings
In the beginning there was ASCII

- Internally every data item in a computer is represented simply by a bit-pattern
- To store integers this is not a problem, because we can simply store their binary representation
- However for non-numerical data such as characters and text we need some sort of encoding that assigns a number (really a bit-pattern) to each character
- In 1968, the American National Standards Institute announced a code called ASCII - the American Standard Code for Information Interchange
  - This was actually an updated version of an earlier code
### ASCII

- ASCII specified numerical codes for 96 printing characters and 32 “control characters” making a total of 128 codes.
- The upper-case alphabetic characters ‘A’ to ‘Z’ were assigned the numerical codes from 65 onwards.

<table>
<thead>
<tr>
<th>A</th>
<th>65</th>
<th>B</th>
<th>66</th>
<th>C</th>
<th>67</th>
<th>D</th>
<th>68</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>69</td>
<td>F</td>
<td>70</td>
<td>G</td>
<td>71</td>
<td>H</td>
<td>72</td>
</tr>
<tr>
<td>I</td>
<td>73</td>
<td>J</td>
<td>74</td>
<td>K</td>
<td>75</td>
<td>L</td>
<td>76</td>
</tr>
<tr>
<td>M</td>
<td>77</td>
<td>N</td>
<td>78</td>
<td>O</td>
<td>79</td>
<td>P</td>
<td>80</td>
</tr>
<tr>
<td>Q</td>
<td>81</td>
<td>R</td>
<td>82</td>
<td>S</td>
<td>83</td>
<td>T</td>
<td>84</td>
</tr>
<tr>
<td>U</td>
<td>85</td>
<td>V</td>
<td>86</td>
<td>W</td>
<td>87</td>
<td>X</td>
<td>88</td>
</tr>
<tr>
<td>Y</td>
<td>89</td>
<td>Z</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© CSSE, Gordon Royle 2008
• The lower-case alphabetic characters ‘a’ to ‘z’ were assigned the numerical codes from 97 onwards

<table>
<thead>
<tr>
<th>a</th>
<th>97</th>
<th>b</th>
<th>98</th>
<th>c</th>
<th>99</th>
<th>d</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>101</td>
<td>f</td>
<td>102</td>
<td>g</td>
<td>103</td>
<td>h</td>
<td>104</td>
</tr>
<tr>
<td>i</td>
<td>105</td>
<td>j</td>
<td>106</td>
<td>k</td>
<td>107</td>
<td>l</td>
<td>108</td>
</tr>
<tr>
<td>m</td>
<td>109</td>
<td>n</td>
<td>110</td>
<td>o</td>
<td>111</td>
<td>p</td>
<td>112</td>
</tr>
<tr>
<td>q</td>
<td>113</td>
<td>r</td>
<td>114</td>
<td>s</td>
<td>115</td>
<td>t</td>
<td>116</td>
</tr>
<tr>
<td>u</td>
<td>117</td>
<td>v</td>
<td>118</td>
<td>w</td>
<td>119</td>
<td>x</td>
<td>120</td>
</tr>
<tr>
<td>y</td>
<td>121</td>
<td>z</td>
<td>122</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Other useful printing characters were assigned a variety of codes, for example the range 58 to 64 was used as follows:

<table>
<thead>
<tr>
<th>:</th>
<th>58</th>
<th>;</th>
<th>59</th>
<th>&lt;</th>
<th>60</th>
<th>=</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>62</td>
<td>?</td>
<td>63</td>
<td>@</td>
<td>64</td>
<td>A</td>
<td>65</td>
</tr>
</tbody>
</table>

As computers became more ubiquitous, the need for additional characters became apparent and ASCII was extended in various different ways to 256 characters.

However any 8-bit code simply cannot cope with many characters from non-English languages.
Unicode

- Unicode is an international code that specifies numerical values for characters from almost every known language, including alphabets such as Braille.
- Java’s `char` type uses 2 bytes to store these Unicode values.
- For the convenience of pre-existing computer programs, Unicode adopted the same codes as ASCII for the characters covered by ASCII.
To characters and back

- To find out the code assigned to a character in Java we can simply cast the character to an int.

- Conversely we can cast an integer back to a char to find out what character is represented by a certain value.
Character Arithmetic

• Using the codes we can do character “arithmetic”
• For example, it is quite legitimate to increment a character variable as in the following code

    char ch;
    ch = ‘A’;
    ch++;

• Now ch has the value ‘B’
Characters as numbers

• As characters are treated internally as numbers, this means they can be freely used in this way

• A loop involving characters
  ```java
  for (char ch = ‘a’; ch <= ‘z’; ch++) {
    // ch takes the values ‘a’ through ‘z’ in turn
  }
  ```

• You can use characters in a `switch` statement
  ```java
  switch (ch) {
    case ‘N’:  // move north
    case ‘E’:  // move east
    case ‘W’:  // move west
    case ‘S’:  // move south
  }
  ```
Unicode notation

• Unicode characters are conventionally expressed in the form
  \[ U+dddd \]

• Here dddd is a 4-digit hexadecimal number which is the code for that character

• We have already seen that ‘A’ is represented by the code 65, which is 41 in hexadecimal

• So the official Unicode code for ‘A’ is U+0041
Unicode characters in Java

• Java has a special syntax to allow you to directly create characters from their U-numbers

```java
char ch;
ch = '\u0041';
```

• You can of course do this in BlueJ’s code pad
More interesting characters

See [www.unicode.org](http://www.unicode.org) for these code charts
Strings

• A string is a sequence of (Unicode) characters
  ABCDEFGHIJ
  Hello, my name is Hal

• One of the major uses of computers is the manipulation and processing of text and so string operations are extremely important

• Java provides support for strings through two classes in the fundamental `java.lang` package: `String` and `StringBuffer`
**String literals**

- You can create a `String` literal just by listing its characters between quotes.

```java
String s = "Hello";
String s = "\u2600\u2601\u2602"
```
Java Programming

java.lang.String

• The class String is used to represent immutable strings
  – Immutable means that a String object cannot be altered after it has been created
• In many other languages a string actually IS just an array of characters, and so it is quite legal to change a single character with commands like
  \[ s[23] = 'z' \]
• There are a variety of reasons for having Strings being immutable including certain aspects of efficiency and security
Methods in the String class

- The `String` class provides a wide variety of methods for creating and using strings.
- Two basic methods are:

```java
public int length()
- this returns the number of characters in the String
public char charAt(int index)
- This returns the character at the given index, where as usual the indexing starts at 0
```
Processing a String

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

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

**compareTo**

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

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns 0 exactly when the `equals(Object)` method would return `true`.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let $k$ be the smallest such index; then the string whose character at position $k$ has the smaller value, as determined by using the `<` operator, lexicographically precedes the other string. In this case, `compareTo` returns the difference of the two character values at position $k$ in the two string -- that is, the value:

```
this.charAt(k) - anotherString.charAt(k)
```

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

```
this.length() - anotherString.length()
```
Lexicographic ordering

• Lexicographic ordering is like alphabetic ordering
• First we order the alphabet
  \[ a, b, c, d, e, f, \ldots, z \]
• The following words are alphabetically ordered
  aardvark, apple, applet, ban, band
• What are the rules for alphabetic ordering of two words?
  – Find the first character where the two words are different and use that character to order the words e.g. aardvark before apple
  – If there are no such characters, then use the length of the words to order them e.g. ban before band
compareTo

• It is the *Unicode value* of the characters that determines their ordering, so for example
  
  *xylophone* comes before *apple*

• The method just specifies that it returns either a negative number, 0, or a positive number as follows:
  
  – A negative number if the target occurs *before* the argument
  – A positive number if the target occurs *after* the argument
  – Zero if the target is equal to the argument
Other methods

- To convert a `String` to lower case:
  ```java
  public String toLowerCase()
  ```

- Hey, I thought Strings were immutable! How can you change it to lower case?
  - I haven’t because this creates a NEW `String` that is a lower-case version of the old one
  - This duplication of `Strings` can be very memory-intensive
Many other methods

```java
public int indexOf(char ch)
public int indexOf(String s)
   – Find the first occurrence in the target string of the character ch, or substring s and return their location

public String replace(char oldChar, char newChar)
   – Create a new String by replacing all occurrences of oldChar with newChar

public char[] toCharArray()
   – Retrieve the characters in the String as an array of chars
```
Concatenation

- We have already seen that the + operator can be used to concatenate strings
  
  String s1 = "Hello";
  String s2 = " there";
  String s = s1 + s2;

- The immutability of Strings can have serious consequences for memory usage that may catch out the unaware - suppose for example that we had to create a single String consisting of all the words in a book
**Slow code**

```java
public String concatenate(String[] words) {
    String text = words[0];
    for (int i=1; i<words.length; i++) {
        text = text + " " + words[i];
    }
    return text;
}
```

This code is disastrously slow if the number of words is even moderately large (a few thousand) because every single time through the loop creates an entirely new `String` with just one word added, hence a vast amount of copying is done and a large number of orphaned objects are created.
**Mutable Strings**

- The class `StringBuffer` is used to represent strings that can be efficiently altered.
- Internally a `StringBuffer` is (essentially) an array of characters.
- It provides efficient ways to *append* and *insert* with a whole range of methods of the following form:

  ```java
  public StringBuffer append(String s)
  public StringBuffer insert(String s, int offset)
  ```

- As of JavaSE 5.0 `StringBuilder`, a single-threaded, non-synchronised class is available as a *drop-in* replacement of `StringBuffer`. 
Appending

public StringBuffer append(String s)
   - Appends the String s to the end of the target StringBuffer
   - It then returns a reference to the newly altered StringBuffer

• Notice that the method both alters the target object and also returns a reference to it

```java
StringBuffer s1 = new StringBuffer("Hello");
s1.append(" there");
```
Using a StringBuffer to concatenate

```java
public String concatenate(String[] words) {

    StringBuffer text = new StringBuffer(words[0]);

    for (int i=1; i<words.length; i++) {
        text.append(" ");
        text.append(words[i]);
    }

    return new String(text);
}
```
**How much difference does it make?**

<table>
<thead>
<tr>
<th>Number of words</th>
<th>Using String</th>
<th>Using StringBuffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.07s</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>0.33s</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>3.29s</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>60.0s</td>
<td>0.01s</td>
</tr>
<tr>
<td>16000</td>
<td></td>
<td>0.02s</td>
</tr>
<tr>
<td>32000</td>
<td></td>
<td>0.14s</td>
</tr>
</tbody>
</table>
Inserting

- A StringBuffer also permits characters or strings to be inserted into the middle of the string it represents.

```java
public StringBuffer insert(int offset, String s)
```

- This inserts the string `s` into the StringBuffer starting at the location `offset` - the other characters are “shifted along”

```
0123456789
Hello John
```
Inserting

```java
StringBuffer s = new StringBuffer("Hello John");
s.insert(5, " to");
```

```
0123456789   01234   56789
Hello John   Hello   John
```

```
0123456789...
Hello to John
```
Inside a StringBuffer

• The StringBuffer internally maintains an array to store the characters
• Usually the array is a bit longer than the number of characters currently stored
• If appends or inserts cause the number of characters to exceed the capacity, then the StringBuffer automatically creates a new bigger array and copies everything over
• This basic mechanism is used in all of Java’s “growable” classes
CITS 1200 Java Programming

Game of Life I
Scope of this lecture

• The Game of Life
• Implementation
• Performance Issues

• References:
  – http://www.bitstorm.org/gameoflife/
  – (and hundreds of other websites)
Game of Life

- Invented by John Conway over 30 years ago
- An example of a “cellular automaton”
- The game is played on a rectangular grid of “cells”
A model world

• This grid is a “model world”, where each cell is either occupied, or vacant
• The initial configuration can be specified by the user, or chosen randomly

We colour the occupied cells, and leave the others blank; the colour scheme is irrelevant to the game, but makes the application look more attractive
Births and Deaths

- In this model world, time passes in discrete steps known as generations.
- The beginning of time is generation zero.
- At each time step, the occupants of the model world live or die according to the following rules:
  - any individual with zero or one neighbours dies of loneliness.
  - any individual with four or more neighbours dies of overcrowding.
  - a new individual is born in any unoccupied cell with exactly three neighbours.
- (The neighbours of a cell are the occupants of the 8 cells surrounding that particular cell.)
Example

Initial configuration

Number of neighbours of each cell

Next generation
The occupants of these cells died of loneliness

These two are new-born “babies”
At the edges

- The edges of the world are assumed to “wrap around” so that the map is a flat map of a torus shaped world.
Program Design

• The program needs classes to perform the following operations
  – Maintaining and updating the map of the world
  – Displaying and updating screen view of the world

• We will design two classes for these two aspects, and use a SimpleCanvas to provide the display

• Therefore we will create two classes:
  – Life
  – LifeViewer
Separate the responsibilities

• The **Life** class should
  – Maintain the current map of the world
  – Update to “next generation” when requested
  – Provide access to the map for client use

• The **LifeViewer** class should
  – Create a **SimpleCanvas** to display the map
  – Display a single generation
  – Animate a client-specified number of generations
The *Life* instance variables

```java
public class Life {
    private boolean[][] map;
    private int width;
    private int height;

    // constructors & methods
}
```

We use a 2d array of type `boolean` for the map of the world; the value of the \([i][j]\) entry in the array `map` indicates whether the \((i,j)\) cell is populated or not.
The first constructor

```java
public Life(boolean[][] initial) {
    map = initial;
    width = map.length;
    height = map[0].length;
}
```

This constructor allows the client to create a `Life` object with any initial pattern of occupied and unoccupied cells that they want.
The second constructor

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

This allows the client to specify just the width and height and have a *random* starting configuration created
Private helper method for initializing

```java
private void initializeMap(double prob) {
    for (int i=0; i<width; i++) {
        for (int j=0; j<height; j++) {
            if (Math.random() < prob) {
                map[i][j] = true;
            } else {
                map[i][j] = false;
            }
        }
    }
}
```

If the user specifies 0.2 as the probability, then approximately 20% of the cells will be occupied.

`Math.random()` is a quick way to get random doubles between 0 and 1.
The default constructor

• It is always polite to give a “default” constructor for users who just wish to get going quickly, and are willing to accept programmer chosen values

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

• Calls the 3-arg constructor with 128, 128 and 0.1 as the values
A new generation

```java
public void nextGeneration() {
    boolean[][] nextMap = new boolean[width][height];
    for (int i=0;i<width;i++) {
        for (int j=0;j<height;j++) {
            int n = numNeighbours(i,j);
            if (n <= 1 || n >= 4) {nextMap[i][j] = false;}
            if (n == 2) {nextMap[i][j] = map[i][j];}
            if (n == 3) {nextMap[i][j] = true;}
        }
    }
    map = nextMap;
}
```
boolean[][] nextMap = new boolean[width][height];

This creates the *space* for the “next generation” to be stored in; we use the instance variables *width* and *height* which are always maintained at the correct values.

We cannot do the updating “in place” because the rules specify that the updating from generation to generation occurs over the entire grid *simultaneously*; we cannot change some values and then use the new values in the calculation for other cells.
```java
int n = numNeighbours(i,j);
if (n <= 1 || n >= 4) {nextMap[i][j] = false;}
if (n == 2) {nextMap[i][j] = map[i][j];}
if (n == 3) {nextMap[i][j] = true;}
```

The main loop processes each cell \((i, j)\) in turn. The number of neighbours is calculated and the \((i, j)\) cell of `nextMap` is set according to the rules of the Game of Life.
Code dissection

```java
map = nextMap;
```

The final statement now makes the instance variable `map` refer to the newly completed “next generation”. Now everything is correctly updated.
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;
}
Here, \( \text{ip} \), \( \text{im} \), \( \text{jp} \) and \( \text{jm} \) refer to \( i+1 \), \( i-1 \), \( j+1 \) and \( j-1 \) respectively.

\[
\begin{array}{ccc}
[i-1][j-1] & [i][j-1] & [i+1][j-1] \\
[i-1][j] & [i][j] & [i+1][j] \\
[i-1][j+1] & [i][j+1] & [i+1][j+1] \\
\end{array}
\]

```java
int ip = (i+1)%width;
int im = (width+i-1)%width;
int jp = (j+1)%height;
int jm = (height+j-1)%height;
```
Dealing with wrap-around

• When \( i \) is equal to \( \text{width}-1 \), then \( i+1 \) is equal to \( \text{width} \), which is off the right-hand side of the picture

• Calculating \( (i+1) \mod \text{width} \) has the effect of “wrapping around” back to 0 which is what is needed

• Similarly, when \( i=0 \), then \( (\text{width}+i-1) \mod \text{width} \) “wraps around” back to \( \text{width}-1 \)
More dissection

```java
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;
```

This is simply a sequence of 8 if statements that just add 1 to the variable n if the corresponding cell is occupied.

Finally the value n is returned.

Notice that the method is private, and only to be used by the Life object itself.
One performance issue

• As written, this code consumes a large amount of memory
What happens during the update?

• After the statement `map = nextMap`
What happens during the update?

- When the method finishes..

This area of memory is now “garbage”
Java will automatically “collect” the garbage for you and recycle it, but there is a performance penalty associated with this.

If the world is large and you are simulating many generations then the memory will very rapidly fill up and the garbage collector will keep interrupting the smooth “animation”.

To fix this, it is better to keep two “maps of the world” as instance variables and then just swap them over at each generation!

```java
private boolean[][] map;
private boolean[][] nextMap;
```
In the `nextGeneration()` method

- Delete the code
  ```java
  boolean[][] nextMap = new boolean[width][height];
  ```
because we do not want to create a new array every generation, but instead re-use the `instance variable` `nextMap`

- Replace
  ```java
  map = nextMap;
  ```
with
  ```java
  boolean[][] swap = map;
  map = nextMap;
  map = nextMap;
  nextMap = swap;
  ```
How does this work?
CITS 1200 Java Programming

Game of Life II
Scope of this lecture

• The Game of Life (cont)
• Display Issues
• Experimentation

• References:
  – http://www.bitstorm.org/gameoflife/
  – (and hundreds of other websites)
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 the LifeViewer interacts with the Life object
  - The internal structure of the LifeViewer object
Interaction

- The LifeViewer needs the Life object to do the following:
  - Send the data for the current generation to be displayed
  - Update to the next generation when required
- This means that the Life class must have the following methods

```java
public boolean[][] getMap()
    • When called, this will return the array representing the current map

public void nextGeneration()
    • When called, this will update the Life object to the next generation
```
The extra method for Life

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

Very simple method; simply returns a reference to the array representing the current generation
Variables defined in LifeViewer

public class LifeViewer {
    private Life life;
    private int width;
    private int height;
    private SimpleCanvas c;

    private final static int CELL_SIZE = 4;

    private final static java.awt.Color BACK_COLOUR = java.awt.Color.white;
    private final static java.awt.Color CELL_COLOUR = java.awt.Color.red;
    private final static java.awt.Color GRID_COLOUR = java.awt.Color.black;
public class LifeViewer {
    private Life life;
    private int width;
    private int height;
    private SimpleCanvas c;

    Each LifeViewer is responsible for displaying one Life object
    The width and height are stored as instance variables for convenience
    Each LifeViewer has one SimpleCanvas on which to draw
Class Variables for LifeViewer

private final static int CELL_SIZE = 4;

private final static java.awt.Color BACK_COLOUR = java.awt.Color.white;
private final static java.awt.Color CELL_COLOUR = java.awt.Color.red;
private final static java.awt.Color GRID_COLOUR = java.awt.Color.lightGray;

Here we define useful constants for the size of the cells and the colours to be used for the drawing. They are declared final because they will not change during the runtime of the program.
The constructor for LifeViewer

```java
public LifeViewer(Life life) {

    this.life = life;
    width = life.getMap().length;
    height = life.getMap()[0].length;

    c = new SimpleCanvas("Life", width*CELL_SIZE+1,
                        height*CELL_SIZE+1, false);

    display();
}
```
What the constructor does

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

The constructor’s job is to initialize all the instance variables; the argument is an instance of the class Life, and so this must be saved in the corresponding instance variable. The width and height variables are then determined by asking the life object for a reference to its 2-d boolean array, and finding its dimensions.

The user must pass the LifeViewer an instance of Life to be displayed.
The constructor for LifeViewer

c = new SimpleCanvas("Life", width*CELL_SIZE+1,
                      height*CELL_SIZE+1, false);

We need the "+1" to give the right hand edge and the bottom edge a nice border
Auto-repaint

- We are using the 4-arg constructor for SimpleCanvas where the fourth argument is a boolean indicating whether the user wants to use automatic repainting or manual repainting.
- Behind the scenes, every drawing command has two effects:
  - the new pixel values are stored by the SimpleCanvas
  - the new pixel values are displayed on the computer screen
- The first operation is very fast, because it just involves the computer’s memory, while the second is much slower.
- When automatic repainting is turned off, drawing commands just affect the memory and the screen is only changed when the client explicitly requests a repaint.
  - can do lots of drawing commands, followed by one single repaint.
Displaying the *Life* object

- Now we need a method to display the *Life* object
- This is basically three steps
  - getting the current map from the *Life* object
  - drawing the appropriate picture on the drawing area, which will involve painting every cell of the grid the right colour
  - calling `repaint()` to have the drawing rendered to the screen
A method to display the Life object

```java
private void display() {
    erase();
    drawCells();
    drawGrid();
    c.repaint();
}
```

Notice how the displaying functions – erasing the old picture, drawing the cells and drawing the grid are all migrated to separate methods; this makes for code that is very easy to read and to update.
Erasing the old picture

private void erase() {
    c.setForegroundColour(BACK_COLOUR);
    for (int i=0; i<width; i++) {
        c.drawLine(i,0,i,height);
    }
}

Familiar code to simply overwrite the entire screen with a collection of vertical lines in the background colour
private void drawCells() {
    boolean[][] map = life.getMap();
    c.setForegroundColour(CELL_COLOUR);
    for (int i=0;i<width;i++) {
        for (int j=0;j<height;j++) {
            if (map[i][j]) {
                for (int k=0;k<CELL_SIZE;k++){
                    c.drawLine(i*CELL_SIZE+k,j*CELL_SIZE,
                    i*CELL_SIZE+k,(j+1)*CELL_SIZE);
                }
            }
        }
    }
}
Drawing the cells

for (int i=0; i<width; i++) {
    for (int j=0; j<height; j++) {
        if (map[i][j]) {
            for (int k=0; k<CELL_SIZE; k++) {
                c.drawLine(i*CELL_SIZE+k, j*CELL_SIZE,
                           i*CELL_SIZE+k, (j+1)*CELL_SIZE);
            }
        }
    }
}
private void drawGrid() {
    c.setForegroundColour(GRID_COLOUR);

    for (int i=0;i<=width;i++) {
        c.drawLine(i*CELL_SIZE,0,i*CELL_SIZE,height*CELL_SIZE);
    }
    for (int j=0;j<=height;j++) {
        c.drawLine(0,j*CELL_SIZE,width*CELL_SIZE,j*CELL_SIZE);
    }
}
A method to display the Life object

```java
private void display() {
    erase();
    drawCells();
    drawGrid();
    c.repaint();
}
```

Finally, we repaint the SimpleCanvas with the new generation drawn on it in order to make all the changes visible on the screen.
public void animate(int n) {
    for (int i=0;i<n;i++) {
        life.nextGeneration();
        display();
        c.wait(250);
    }
}

The only public method is the one that allows the user to specify that they want to see n generations of “The Game of Life” one after the other; it is a simple loop that asks the Life object to calculate the next generation, then displays it, and then waits briefly to give an “animated” effect.
Create the *Life* object
It appears on the workbench
Now create a **LifeViewer** to view it
The initial randomly chosen configuration is displayed on a SimpleCanvas.
After 100 generations, the map looks a bit more “organized”
A new colour scheme can be obtained just by changing the three constants.
Extensions

• There are hundreds of research papers, websites and recreational articles about Conway’s Game of Life
• Life is just one example of an entire class of systems known as cellular automata, many of which have been heavily studied
• There are many variants, but we will just consider what happens if we change the rules determining birth and death
• Conway’s rules are sometimes described as the String $B3/S23$
  – a new individual is born in a cell with 3 neighbours
  – an individual survives in a cell with 2 or 3 neighbours
Extending the code

- We could simply alter the code to accommodate any particular set of rules, but a more object-oriented approach would be to devise a class so that any such rule set could be “plugged in” by the client.
- This maximizes the re-use of the existing code, and minimizes the temptation to “cut-n-paste” every time the user wants to experiment with a new rule set.
- However, this does require re-engineering the Life class.
The CARule interface

- The CARule class has the following *interface* - that is, the following list of methods

- The implementation of the methods isn’t yet specified

```java
public class CARule {
    public CARule(String b, String s);
    public boolean birth(int value);
    public boolean survival(int value);
}
```

The role of a CARule object is simply to respond as to whether a particular value is a “birth” value or not, or a “survival” value or not
The new Life class

• The Life class will now maintain a CARule object as an instance variable car, and refer to it whenever it wants to decide whether a particular value should cause an individual to be born or survive

```java
for (int i=0; i<width; i++) {
    for (int j=0; j<height; j++) {
        int n = numNeighbours(i, j);
        nextMap[i][j] = false;
        if (map[i][j] && car.survival(n)
            || !map[i][j] && car.birth(n))
            nextMap[i][j] = true;
    }
}
```
CITS 1200 Java Programming

Defensive Programming
Scope of this lecture

• Why program defensively?
• Encapsulation
• Access Restrictions
• Documentation
• Unchecked Exceptions
• Checked Exceptions
• Assertions
Why program defensively?

- Normally, your classes will form part of a larger system, so other programmers will be using and relying upon your classes.
- Obviously, your classes should be correct, but equally importantly, your classes should be robust – resistant to 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.
- We use the terminology “client code” to indicate the code written by other programmers that is using your classes and methods.
Encapsulation

- One of the most important features of OOP is that it facilitates *encapsulation* – a class encapsulates both the data it uses, and the methods to manipulate the data.
- The external user *only* sees the public methods of the class, and interacts with the objects of that class purely by calling those methods.
- This has several benefits:
  - Users are insulated from needing to learn details outside their scope of competence.
  - Programmers can alter or improve the implementation without affecting any client code.
Access Restrictions

• Encapsulation is enforced by the correct use of the access modifiers, public, private, <default>, and protected

• If you omit the access modifier, then you get the default which is sometimes known as “package”

• These latter two modifiers are only really relevant for multi-package programs that use inheritance, so we need only consider public and private at the moment
**public and private**

- If an instance variable is **public**, then
  - Any object can *access* it directly
  - Any object can *alter* it directly

- If an instance variable is **private**, then
  - Objects that belong to *the same class* can access and alter it
  - Notice that privacy is a per-class attribute not per-object

- If a method is **public**, then
  - Any object can call that method

- If a method is **private**, then
  - Objects that belong to *the same class* can call it
Public methods

• The public interface of a class is its list of public methods, which details all of the services that this class provides.

• Once a class is released (for example, as part of a library), then it is impossible or very difficult to change its public interface, because client code may use any of the public methods.

• Public methods must be precisely documented and robust to incorrect input and accidental misuse.

• 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 variables

• Normally instance variables should not be public
• Client code can directly access and/or alter the values of public instance variables, and client programmers must be aware of the internal structure of the class
  – Benefit of encapsulation is lost when instance variables are public
• If client access to instance variables is desirable, then it should be provided by accessor and/or mutator methods (getters and setters)
• Two important benefits of this approach
  – Maintenance of object integrity
  – Permits change of implementation
Simple Example

class MyDate {
    public int day;
    public String month;
    public int year;
}

MyDate md = new MyDate();
md.day = 31;
md.month = "Feb";

Now md is corrupt and will cause problems elsewhere in the program!
public void setDate(int day, String month) {
    // Check that the month is sensible, and that the
    // date is valid before setting the variables
}

The method acts as a “gatekeeper” and rejects values that would create a corrupt object
Documentation

• Documentation should be developed *at the same time*, and *in the same place* as the source code
  – A very common occurrence is that documentation is put off until the end of the project, and often it is never written

• Java provides facilities to help with this, by allowing source code comments to be automatically turned into documentation using the *javadoc* feature

*The documentation is a contract between the programmers and users of a class!*
**javadoc**

- Comments occurring between the special tags /*** and */ are treated as special javadoc comments
- These can be automatically parsed (processed) to generate HTML documentation that can be read by a Web browser
- The resulting documentation is used to precisely define the behaviour of the class and its methods
- The Java API itself is generated from source code comments by javadoc
Javadoc comments

• Place comments directly before a class, field (i.e. variable),
  constructor or method declaration
• Critically important for public classes, fields, constructors
  and methods, but may also be used for private ones if
  desired
• Comments can be written in HTML (the code used for
  Web pages), and enhanced with special javadoc tags
  such as

  @author, @version
  @param, @return, @throws
  @see
javadoc comments

```java
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.getContentPane().add(canvas);
    frame.pack();
    frame.setVisible(true);
```
Uncheck this if offline, or it will try to link to Sun’s Java documentation.
Generate Documentation

```
import java.awt.Canvas;

public class SimpleCanvas extends Canvas {
    public SimpleCanvas(String title, int width, int height, boolean autoscale) {
        super(title, width, height, autoscale);
    }
}
```

**Parameters:**
- `title`: title for the window
- `width`: the desired width of the SimpleCanvas
- `height`: the desired height of the SimpleCanvas
- `autoscale`: true for automatic rescaling

**Method Detail**
Dealing with Errors

• Even if your classes are well-protected, errors still occur
• We consider three different types of error
  – Client code attempts to use your methods incorrectly, by passing incorrect or invalid parameter values
  – Your code cannot perform the services it is meant to due to circumstances outside your control (such as an Internet site being unavailable)
  – Your own code behaves incorrectly and/or your objects become corrupted
• Java provides three mechanisms to handle these types of situation – *unchecked exceptions*, *checked exceptions* and *assertions*
Invalid parameters

- The method `charAt(int index)` returns the character at the specified index in a `String`.
- The only *valid* values for the parameter are numbers from 0 up to one less than the length of the `String`.
- What happens if `charAt(-1)` is ever called?
The method “throws” an \textit{Exception}

- If a parameter is invalid, then the method cannot do anything sensible with the request and so it creates an object from an \texttt{Exception} class and “throws” it.

- If an \texttt{Exception} is thrown, then the runtime environment immediately tries to deal with it:
  - If it is an \textit{unchecked} exception, it simply causes the runtime to halt with an error message.
  - If it is a \textit{checked} exception, then the runtime tries to find some object willing to deal with it.

- The method \texttt{charAt} throws a \texttt{StringIndexOutOfBoundsException} which is unchecked and hence causes the program to cease executing.
Throw your own

- Your own methods and/or constructors can throw exceptions if clients attempt to call them incorrectly.
- This is how your code can enforce rules about how methods should be used.
- For example, we can insist that the `deposit` and `withdraw` methods from the `BankAccount` class are called with positive values for the amount.
- The general mechanism is to check the parameters and if they are invalid in some way to then:
  - Create an object from class `IllegalArgumentException`
  - Throw that object.
public void deposit(int amount) {
    if (amount <= 0) {
        IllegalArgumentException ie;
        ie = new IllegalArgumentException("Negative Amount");
        throw(ie);
    } else {
        balance = balance + amount;
        // the other work
    }
}

• If the amount is negative then declare the variable ie, create the object and then throw it

• The constructor for IllegalArgumentException takes a String argument which is used for an error message that is returned to the user
Constructors too...

- Throwing an exception is also how a constructor can prohibit the construction of invalid objects.
- The RGB-constructor for `java.awt.Color` will throw an `IllegalArgumentException` if the user tries to create an illegal colour:
  
  ```java
  new java.awt.Color(300,120,-5);
  ```
“Predictable” Errors

- Unchecked exceptions terminate program execution and are used when the client code must be seriously wrong.
- However, there are error situations that do not necessarily mean that the client code is incorrect, but reflect either a transient, predictable or easily-correctable mistake – this is particularly common when handling end-user input, or dealing with the operating system.
- For example, printers may be out of paper, disks may be full, Web sites may be inaccessible, filenames might be mistyped and so on.
Checked Exceptions

• Methods prone to such errors may elect to throw *checked* exceptions, rather than unchecked exceptions
• Using checked exceptions is more complicated than using unchecked exceptions in two ways:
  – The programmer must *declare* that the method might throw a checked exception, and
  – The client code using that method is *required* to provide code that will be run if the method *does* throw an exception
The client perspective

• Many of the Java library classes declare that they might throw a checked exception
try and catch

• If code *uses* a method that might throw a checked exception, then it *must* enclose it in a `try/catch` block

```java
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
try and catch continued

- If everything goes smoothly, then the code in the `try` block is executed and the code in the `catch` block is skipped.
- Otherwise, if one of the statements in the `try` block causes an exception to be thrown, then execution immediately jumps to the `catch` block, which tries to recover from the problem.
- What can the `catch` block do?
  - If dealing with a human user, the `catch` block may report the error and ask the user to change their request, or retype their password, or specify another URL etc.
  - In any situation, the `catch` block should provide some feedback as to the likely cause of the error and how it may be overcome, even if it ultimately it just causes execution to cease.
Advanced `try/catch` features

- Other features of `try/catch` that are beyond the scope of this unit include
  - The ability to include several `catch` blocks to deal with different possible types of exception
  - The ability to include a statement block that is always run after all the other blocks have been given a chance to run (`finally`)
  - The ability to “give up” and re-throw the exception (with or without attempting to deal with it)

- Java provides a considerable number of 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
The programmer perspective

• If you choose to write a method that throws a checked exception, then this must be declared in the source code, where you must specify the type of exception that might be thrown.

```java
public void printFile(String fileName) throws java.io.FileNotFoundException {
    // Code that attempts to print the file
}
```

• If you declare that your method might throw a checked exception, then the compiler will force any client code that uses your method to use a try/catch block.
  – This explicitly makes the client code responsible for these situations, hence the client programmers must consider this possibility.
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 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 checked exceptions only if you believe that the client code might be able to do something about the problem!
Assertions

• Assertions are a debugging mechanism for the programmer to use when developing complicated code
• At any point, the programmer can add a line of code with the following structure

    assert <boolean-condition> : <string>;

• When the assertion is reached, the boolean condition is evaluated and if it is false then execution is halted with an AssertionError (a type of unchecked exception) and the message string is printed
Why use assertions?

• Assertions are used to help track down logic errors
• As you program a complicated piece of code, you mentally keep track of what a given variable *should* or *could* contain
• Using an assertion allows you to express formally what you *expect* to be the case, and have this checked during runtime
• Without assertions an error may appear a long way from the code that actually caused it
  – In this situation, the root cause of the bug is not apparent and the error message may be very cryptic
CITS 1200 Java Programming

Unit Testing and JUnit
Eschew Obfuscation

Suppose we wanted to make lab and project sheets really hard for students to read and use – what should we do?

List as many ways as you can think of to do this
Lecture Outline

• Communicating the Requirements for Java programs

• Introduction to JUnit4

• How to read and write test cases
Prose Specification

- Very flexible
- May not match code
- Long and wordy – time consuming to read and write
- Little structure – hard to find info
- May be inconsistent
UML Class Diagrams

- Give an external view of a class
- But there are lots of questions they don’t answer
- If you want to use this class, you will need more information
- For example, what does the resize method do? What do its parameters mean?
- Note that good choice of identified names can help a lot

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>height : int</td>
</tr>
<tr>
<td>width : int</td>
</tr>
<tr>
<td>resize(double, double)</td>
</tr>
<tr>
<td>getArea() : int</td>
</tr>
<tr>
<td>getPerimeter() : int</td>
</tr>
</tbody>
</table>
• Method signature

        public void resize( double hscale, double wscale)

• JavaDoc

        /**
         * Grow or shrink a rectangle in two dimensions.
         *
         * @param hscale double multiplier for height
         * @param wscale double multiplier for width
         */
(Even) More information

• More JavaDoc

/**
 * Grow or shrink a rectangle in both dimensions
 * scaling factors are floats (can be less than 1)
 * but final dimensions are recast as integers.
 * @param hscale the float multiplier for height.
 * @param wscale the float multiplier for width.
 * @throws exceptions for what cases ??
 */
Javadoc

- Structured document
- Standard format
- Easy to navigate html links
- Matches the code … probably
- Only as good as the prose comments
## Learn by Example

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
</table>
| height=3, width=4  
hscale=2.0, wscale=0.5 | height=6, width=2 |
| height=3, width=4  
hscale=1.0, wscale=1.0 | height=3, width=4 |
| height=3, width=4  
hscale=0.5, wscale=0.0 | height=1?, width=0? |
How much info do you need?

• Method signatures don’t tell you \textit{what} the method does

• JavaDoc \textit{can} specify anything you need to know, but we don’t have a way of checking that the code meets the specification

• \textbf{Examples} are helpful for programmers

• Most importantly, we can \textit{check} whether code behaves as the examples claim

• When such checks are themselves Java code, then we can \textit{recheck} that code still works after any change, by \textit{running} the test code
Lecture Outline

- Communicating the Requirements for Java programs
- Introduction to JUnit4
- How to read and write test cases
JUnit

- JUnit is a regression testing framework written by Erich Gamma and Kent Beck. It is used by the developer who implements unit tests in Java.
- JUnit is Open Source Software, released under the Common Public License Version 1.0 and hosted on SourceForge.
- There are JUnit plug-ins for most development environments (IDEs) and XUnit for other languages (C, C++ …)
  - Patrick Doran-Wu at UWA has developed a JUnit4 version of BlueJ
  - Eclipse supports the latest JUnit4
JUnit

- de facto standard unit testing framework
- bundled in most development environments (IDEs)
- www.junit.org
- automatic quality assurance: when the code changes, test cases are re-run to check that nothing breaks
What’s in a test case?

Unit test cases have the following form:

1. describe the method’s inputs
2. describe the expected outputs
3. call the method and observe actual output
4. compare expected with actual and report
Coding a test case in JUnit4

```java
@Test
public void testGetAreaNormal() {

    //1. describe input
    Rectangle r1 = new Rectangle(3, 4);

    //2. describe expected output(s)
    int expected = 12;

    //3. call the method
    int actual = r1.getArea();

    //4. compare actual output with expected
    assertEquals(expected, actual);
}
```
Before *every* test case

```java
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

/**
 * The test class BankAccountBasicTest.
 * test for basic functionality of the BankAccount class
 *
 * @author Rachel Cardell-Oliver
 * @version February 2011
 */
public class BankAccountBasicTest {
    private BankAccount normalAccount, emptyAccount;

    /**
     * set up sample accounts to use for the tests
     */
    @Before public void setUp() {
        normalAccount = new BankAccount("John", 1000);
        emptyAccount = new BankAccount("Jane", 0);
    }
}
```
How a test suite is executed

1. setUp method is executed
2. the first test method in the class is executed
3. tearDown method is executed
4. setUp method is executed
5. the second test method in the class is executed
6. tearDown method is executed

Continue until all test cases are executed.

IMPORTANT each test case is independent of previous tests
Does expected = actual?

To compare any variables of the 8 Java base types (boolean, byte, short, int, long, char, float, double)

`assertEquals(expected, actual)`

First argument (optional) is a string that's printed if the assertion fails:

`assertEquals(message, expected, actual)`
Straight-forward comparisons for basic types

```java
assertEquals(int expected, int actual)
assertEquals(boolean expected, boolean actual)
assertEquals(byte expected, byte actual)
assertEquals(char expected, char actual)
assertEquals(short expected, short actual)
assertEquals(long expected, long actual)
```
Floating point arithmetic

As always with floating point arithmetic (double or float), you want to avoid direct equality comparisons.

```java
public static void assertEquals(double expected, double actual, double tolerance)
```

For example,

```java
assertEquals(0.552, 0.510, 0.1) is true but
assertEquals(0.552, 0.510, 0.01) is false
```

since the values are equivalent within a tolerance of ± 0.1 but not to 2 decimal places.
Array Equality

JUnit4 provides test case for arrays

```java
public static void assertArrayEquals(
        int[] expecteds,
        int[] actuals)
```

Asserts that two int arrays are equal. If they are not an AssertionError is thrown (the test fails)
Can be used with arrays of all basic types (char, short etc) but not double
Testing Array Equality

```java
@Test public void testarrays1() {
    int[] a1 = {1,2,3};
    int[] a2 = {1,2,3};
    assertArrayEquals(a1,a2);
}
```
@Test
    public void testarrays2() {
        int[] a1 = {1,2,3};
        int[] a3 = {1,2,3,4};
        assertArrayEquals(a1,a3);
    }
```java
@Test public void testarrays3() {
    char[] a4 = {'a','b'};
    char[] a5 = {'c','d'};
    assertArrayEquals(a4,a5);
}
```
How to Test Object Equality

What is the expected result of this test?

```java
@Test public void testObjEquality() {
    BankAccount b1 = new BankAccount("Rachel",10);
    BankAccount b2 = new Rectangle("Rachel",10);
    assertEquals(b1,b2);
}
```
Problem: we tested for equivalence of the references to the 2 BankAccount objects, not the contents of those objects
Solution: if you need to do this then write a helper method to check the equality of each field value
Testing bad methods

A test that passes because the expected exception is thrown

```java
@Test(expected=ArithmeticException.class)
public void testdivideByZero() {
    int n = 2 / 0;
}
```

A test that fails with a timeout because this loop never stops

```java
@Test(timeout=1000)
public void testneverStops() {
    for (int i=0; i<10; i=i) { int j=0; }
}
```
Lecture Outline

• Communicating about Java programs

• Introduction to JUnit4

• How to read and write test cases

“Testing can show the presence of bugs but never their absence”
E.W. Dijkstra
How Test Cases are Chosen

• Typical cases
  – Sanity check that method code works as expected, but try to choose tricky cases anyway!

• Special cases and Boundary cases
  – Most errors occur for boundary values, eg empty or full array, -1,0,1 etc.

• Exceptional cases
  – Illegal input, divide by 0, un-initialized parameters
Plan your test cases using tables

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPICAL</td>
<td></td>
</tr>
<tr>
<td>height=3, width=4</td>
<td>height=6, width=2</td>
</tr>
<tr>
<td>hscale=2.0, wscale=0.5</td>
<td></td>
</tr>
<tr>
<td>BOUNDARY</td>
<td></td>
</tr>
<tr>
<td>height=3, width=4</td>
<td>height=3, width=4</td>
</tr>
<tr>
<td>hscale=1.0, wscale=1.0</td>
<td></td>
</tr>
<tr>
<td>EXCEPTIONAL</td>
<td></td>
</tr>
<tr>
<td>height=3, width=4</td>
<td>height=1?, width=0?</td>
</tr>
<tr>
<td>hscale=0.5, wscale=0.0</td>
<td></td>
</tr>
</tbody>
</table>
Also, ask questions

- Useful when testing someone else’s class eg. a library class
- Ask: What happens if you do XXX?
  - Is an exception thrown? If so, which one?
  - Is null treated as zero?
  - Is null returned?
  - Something else?
- Write a test that determines the answer to your question(s)
More ways to test your code

Acceptance Tests
- Use a `main` method to execute a sequence of steps (method calls) corresponding to client’s use cases
- Good for visual testing of methods with graphical displays
- The TicTacToeDemo is a good example

Coverage Testing
- How much of your code is executed by your test cases
- For example, try to execute all branches
More ways to test your code

Debugging

• Sometimes `println` statements are useful to pin down where (and why) the code is failing – for example in the middle of a method.

• Also use the BlueJ debugger to stop and inspect your code
CITS 1200 Java Programming

Searching and Sorting I
Scope of this lecture

- Data Arrays
- Searching and Linear Search
- Sorting and Bubble Sort

References:
- Wirth, *Algorithms + Data Structures = Programs*, Chapter 2
Data Arrays

- Computers are particularly good at maintaining and manipulating vast quantities of data.
- For example, the marks for CITS1200 form a collection of data items, where each data item is an integer between 0 and 100.
- A data structure is a software construct used to store and provide convenient access to a collection of data.
- The simplest data structure – and the only one we consider in CITS1200 – is the array. The 2nd year unit “Data Structures” covers many more.
- We can consider arrays of primitive types or (more usually) arrays of objects.
Data Manipulation

- Given an array of data items, there are many ways in which we might want to manipulate this data
- We have already considered some ways to perform various statistical analyses of the data items in an array
  - Finding the maximum and minimum elements
  - Finding the mean and standard deviation of the elements
  - Visualizing the distribution of the data
- Two of the most important things that we do with data are searching and sorting
Searching

- *Searching* refers to the process of finding data items that match certain criteria; we may either want a yes/no answer to a question or additional details as well
  - Find out *whether* any students got a mark of 49
  - Find out *which* students got a mark of 49

- The simplest searching technique is called *linear search*, which simply involves looking through each element in turn until we find one that matches the criteria
/* This method returns true if the array a contains a data item equal to value */

public boolean linearSearch(int[] a, int value) {

    for (int i=0; i<a.length; i++) {
        if (a[i] == value) {
            return true;
        }
    }

    return false;
}
Comments

- If the array *does* contain the desired value, then the method returns `true` as soon as it is found.
- If the array *does not* contain the desired value, then the method will return `false` after checking every element of the array without success.
- This situation – finding a number in an array of numbers – illustrates the *principle* of linear search, but is not very realistic, as we would normally need more information than just the fact that a numerical value is in an array.
- In particular, the more general situation is finding an *object* in an array of objects.
Our favourite Student class

public class Student {
    private String studentID;
    private int mark;

    public Student(String studentID, int mark) {
        this.studentID = studentID;
        this.mark = mark;
    }

    public String getStudentID() {
        return studentID;
    }

    public int getMark() {
        return mark;
    }
}

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

- We consider a class list being stored as a `Student[]`
- The question that we will consider is how to retrieve the data for a student with a given student number

- So, we will write a method with the following signature

```java
public Student findStudent(Student[] classlist, String id)
```

- The method returns a (reference to a) `Student` object
- The array of students is a parameter
- The student ID we want is the other parameter
public Student findStudent(Student[] classlist, String id) {

    for (int i=0;i classlist.length;i++) {
        String sid = classlist[i].getStudentID();
        if (sid.equals(id)) {
            return classlist[i];
        }
    }

    return null;
}
Dissection of this code

• The basic structure of the code is simply a linear search, but there are some important nuances
• The main loop looks through each element of the array in turn – this is the fundamental linear search

```java
for (int i=0; i<classList.length; i++) {
    // process the i-th element of the array
}
```
Processing each element

• When we process the \(i\)th element of the array we need to
  – Find out the user id of that student
  – See if it matches the one we are looking for

• Finding out the student id of the \(i\)th element is done by using its “getter” method
  \[
  \text{String sid = classlist[i].getStudentID();}
  \]

• To check whether the student number \(\text{snum}\) is equal to the one we are looking for, we use the \text{equals()} method of the \text{String} class
  \[
  \text{if (sid.equals(id)) {}
    \text{ return classlist[i];}
  \}
  \]
Why the `equals()` method?

- Why must we use
  
  ```java
  if (sid.equals(id))
  ```
  
  instead of
  
  ```java
  if (sid == id)
  ```

- *Answer*: Because *Strings* are objects (not primitive types), and so `sid` and `id` are *references* referencing those objects. Even if `sid` and `id` have the same *value*, they may not be the same object!

- How do we know about the `equals()` method?
  - From browsing the class libraries!
What if we don’t find the Student?

- In the “plain vanilla” linear search, we simply returned `false` if we did not find the value in the array.
- What should the method `findStudent()` return if the student is not in the array?
- There are a number of options, depending on whether this should be viewed as a critical error or not.
- In this case, we assume that it is not a critical error, and it simply reflects the fact that the student is not in this particular group (class, lab, tute etc.), so we take the simplest option.

```java
return null;
```

Recall that `null` is a special value (reference) meaning “not referring to any object.”
Performance of linear search

• How fast does linear search work on an array of \( n \) items?
• We can identify three situations
  – Best case, when the input is the most convenient possible
  – Worst case, when the input is the least convenient possible
  – Average case, averaged over all the inputs
• In the *best* case, linear search finds the item at the first position of the array, so it needs 1 comparison
• In the *worst* case, linear search does not find the item and so must perform \( n \) comparisons unsuccessfully
• To calculate the *average* case performance we would need some problem-specific assumptions about the input data
Linear search is too slow

- If we have very large amounts of data, then linear search is not feasible
- For example, we can view the telephone directory as a very large array of objects, with each object consisting of a name and a number
  - If you are asked to find out which person has phone number 9388 6105 then how long would it take you to do this by linear search?
- However, if I ask you to find out the phone number of a specific person, then you can do it much, much faster
  - How do you do it?
  - How can we program a computer to do this?
Sorted Arrays

- The reason that
  
  ![Diagram](Diagram)

  is quick, while

  ![Diagram](Diagram)

  is slow is because the “array” (i.e. phone book) is sorted into alphabetical order, and somehow this allows us to find an entry much more quickly (we will see why later)

- Most useful databases are sorted – dictionaries, indices etc.
Sorting

• Before we examine how to efficiently search in a sorted array, we must consider how to sort the array.

• We will again start with the “plain vanilla” example – sorting an array of integers into increasing order:

  before  6 8 1 15 12 2 7 4
  after  1 2 4 6 7 8 12 15

• Later we will extend this to sorting arrays of objects according to various other criteria (alphabetical etc).
The basic set up

- We will implement a number of sorting methods, all of which operate on an array of integers.
- We develop these as a utility class called `Sorter` – a class with no instance variables, but just class (static) methods (e.g. `Math`)
- Each method will have a similar signature, where the only thing that will vary is the name of the sorting technique

```java
public static void sortName(int[] a)
```

- Each method receives a (reference to an) array as a parameter and will sort that array in place
Bubblesort

- The idea behind **Bubblesort** is to systematically compare pairs of elements, exchanging them if they are out of order
- If the array contains $n$ elements, then we view the algorithm as consisting of $n$ “passes”
- In the first pass we compare
  - Element 0 with element 1, exchange if necessary
  - Element 1 with element 2, exchange if necessary
  - ... 
  - Element $n-2$ with element $n-1$, exchange if necessary
The first pass

- After the first pass, the largest element will be at the end
The second pass

- The second pass doesn’t need to make the last comparison
The third pass

• The third pass can omit the last two comparisons
The fourth pass

- The fourth pass is even shorter
The fifth and sixth passes

1 2 6 4 7 8 12 15

1 2 6 4 7 8 12 15

1 2 4 6 7 8 12 15

1 2 4 6 7 8 12 15

1 2 4 6 7 8 12 15
Why does it work?

- We need to have some argument or “proof” that this works
- We claim that

  After the i-th pass, the last i elements in the array are in their correct positions

- This is true after the first pass, because the largest element in the array is encountered at some stage and then “swapped all the way to the end” of the array
- The same argument – applied to the remainder of the array – shows that the second pass puts the 2\textsuperscript{nd} largest element into place; repeating this argument \(i\) times gives the result
Coding Bubblesort

public static void bubbleSort(int[] a) {
    for (int pass=1; pass<a.length; pass++) {
        for (int j=0; j<a.length-pass; j++) {
            if (a[j] > a[j+1]) {
                int swap = a[j+1];
                a[j+1] = a[j];
                a[j] = swap;
            }
        }
    }
}
public static void bubbleSort(Student[] a) {
    for (int pass=1; pass<a.length; pass++) {
        for (int j=0; j<a.length-pass; j++) {
            if (/* a[j] and a[j+1] out of order */) {
                Student swap = a[j+1];
                a[j+1] = a[j];
                a[j] = swap;
            }
        }
    }
}
What order do we want?

- The precise form of the statement depends on whether we want to sort students:
  - *Alphabetically* according to their `studentId`
  - *Numerically* according to their `mark`
- In addition, the desired sort could be *ascending* (lower values first) or *descending* (lower values last)
- Suppose that we want to sort the students into normal (ascending) alphabetical order by `studentId`
For alphabetic order

- The comparison between the two Student objects \( a[j] \) and \( a[j+1] \) first needs to obtain the two ids to compare, so it will involve the two Strings
  
  - String \( s1 = a[j].\text{getStudentID}() \);
  
  - String \( s2 = a[j+1].\text{getStudentID}() \);

- To compare two Strings we use the `compareTo` method

        if (s1.compareTo(s2) > 0) {
            // Swap the two Students
        }

© CSSE, Gordon Royle 2008
CITS 1200 Java Programming

Searching and Sorting II
Scope of this lecture

• Insertion Sort
• Timings
• Binary Search

• References:
  – Wirth, Algorithms + Data Structures = Programs
  – Knuth, The Art of Computer Programming, Volume 3, Sorting and Searching
Sorting cards

- In card games, it is common to pick up the cards as they are dealt and to sort them into order as they arrive.
- For example, suppose your first three cards are:

![Card Image]

- Next you pick up a 9 of clubs.
Inserting a card

• The new card is then *inserted* into the correct position
Insertion sort

• We can develop this notion into a sorting algorithm called *Insertion Sort*

• When sorting *n* items, *Insertion Sort* has the following properties:
  – The procedure has *n* stages
  – After the *i*’th stage, the first *i* items are sorted in order
  – At the (*i+1*)’st stage, the item originally in position *i+1* is incorporated into the sorted list and placed in position
Example

• Initial array

| 6 | 8 | 1 | 15 | 12 | 2 | 7 | 4 |

• Stage 1: Move the first element into position (do nothing)

| 6 | 8 | 1 | 15 | 12 | 2 | 7 | 4 |

• Stage 2: Examine the second element and insert it into position (again do nothing)

| 6 | 8 | 1 | 15 | 12 | 2 | 7 | 4 |
Stage 3

- This element is out of position so will have to be *inserted*

```
6 8 1 15 12 2 7 4
```

```
6 8 → 15 12 2 7 4
```

```
6 → 8 15 12 2 7 4
```

```
6 8 15 12 2 7 4
```

```
1 6 8 15 12 2 7 4
```
Stages 4 and 5

• Stage 4

1 6 8 15 12 2 7 4

• Stage 5

1 6 8 15 12 2 7 4

1 6 8 15 2 7 4 12

1 6 8 15 2 7 4 12

1 6 8 12 15 2 7 4
Stage 6

1 6 8 12 15 2 7 4

1 6 8 12 15 7 4

1 6 8 12 15 7 4

1 6 8 12 15 7 4

1 6 8 12 15 7 4

1 6 8 12 15 7 4

1 6 8 12 15 7 4

1 2 6 8 12 15 7 4
Stage 7

1 2 6 8 12 15 7 4

1 2 6 8 12 15 4

1 2 6 8 12 15 4

1 2 6 8 12 15 4

1 2 6 8 12 15 4

1 2 6 8 12 15 4

1 2 6 8 12 15 4
Final stage
Code for InsertionSort

```java
public static void insertionSort(int[] a) {

    for (int pass=1; pass<a.length; pass++) {
        int tmp = a[pass];
        int pos = pass-1;

        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }

        a[pos+1] = tmp;
    }
}
```
Code Dissection

public static void insertionSort(int[] a) {

    for (int pass=1;pass<a.length;pass++) {

        int tmp = a[pass];
        int pos = pass-1;

        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }

        a[pos+1] = tmp;
    }
}
public static void insertionSort(int[] a) {
    for (int pass=1; pass<a.length; pass++) {
        int tmp = a[pass];
        int pos = pass-1;
        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }
        a[pos+1] = tmp;
    }
}

The variable tmp stores the value that is to be inserted; the variable pos will eventually contain the position where it should be inserted.
Code dissection

```
public static void insertionSort(int[] a) {

    for (int pass=1;pass<a.length;pass++) {
        int tmp = a[pass];
        int pos = pass-1;

        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }
        a[pos+1] = tmp;
    }
}
```

This code does the work of shifting each element in turn one space along if it is bigger than the value to be inserted. We also need to ensure that we don’t fall off the left-hand end of the array!
public static void insertionSort(int[] a) {

    for (int pass=1; pass<a.length; pass++) {
        int tmp = a[pass];
        int pos = pass - 1;
        while (pos >= 0 && a[pos] > tmp) {
            a[pos+1] = a[pos];
            pos--;
        }
        a[pos+1] = tmp;
    }
}

The while loop finishes when we have found the correct position for the element, and so it is now inserted into this position.
Performance Testing

- We would like to do some performance testing to see how these algorithms perform under test conditions

> Warning: the test methodology outlined here is primitive and should only be taken as broadly indicative of empirical program testing

- We will test the algorithms by giving them both the same collection of randomly chosen data sets to sort, and comparing how long they take
- We will use the method `System.currentTimeMillis()` to get the current time before and after the sorting process
Java Programming

Testing procedure
public static void timeTest() {
long before;
long after;

Create and duplicate a
random array to be sorted

for (int i=500; i<=5000; i+=500) {
int[] data1 = Sorter.randomArray(i);
int[] data2 = Sorter.duplicate(data1);
before = System.currentTimeMillis();
Sorter.bubbleSort(data1);
after = System.currentTimeMillis();
System.out.print(i+" "+(after-before)+" ");
before = System.currentTimeMillis();
Sorter.insertionSort(data2);

Find the time before,
execute the sort and
find the time after!

after = System.currentTimeMillis();
System.out.println(after-before);
}
}
© CSSE, Gordon Royle 2008

18


Warnings

• `System.currentTimeMillis()` just reports the current elapsed time, not the CPU time used by the current process
  – This means that other activity on the computer will make your program appear artificially slow
  – Many things are usually happening in the background and so even if the computer appears to have nothing else running, this may not be the case
• It is best to run your timing programs several times over and average the results to eliminate these artefacts
• Don’t trust very small times – you should repeat things to get fairly large elapsed times and divide by the number of repeats
• You should ensure that the data you test on is representative
Two sets of results

• The timings were run for **BubbleSort** and **InsertionSort** for randomly generated arrays of length 500-5000
Evaluating the results

Sorting Comparison

- Bubble
- Insertion
Experimental Conclusion

• It seems clear that **InsertionSort** is superior to **BubbleSort** for random data arrays of sizes from 500-5000

• The difference between **InsertionSort** and **BubbleSort** seems to be getting larger and larger as the array size increases

• Is this really true? Or just an illusion caused by the seemingly rapidly rising lines on the graph?

• We get a better view of the “rate of growth” of the function by looking at a plot on a *logarithmic scale*
  – The *slope* of a function on a logarithmic scale is a measure of the rate of growth of that function
Both have same “rate of growth”

Sorting Comparison

Array Size

Time (millis)

Bubble
Insertion

© CSSE, Gordon Royle 2008
Searching

• We return now to the original reason for sorting data – to facilitate searching for specific values
• The linear search technique was too slow for large data sets as it could take as many as \( n \) steps for an array of \( n \) items
• We will consider a technique called binary search (which is somewhat reminiscent of the bisection method)
• As an example, we will consider checking to see if the value 90 occurs in the sorted array

\[
\begin{array}{cccccccccccc}
11 & 18 & 24 & 35 & 49 & 53 & 57 & 62 & 71 & 77 & 82 & 87 & 90 & 92 & 95 & 98 \\
\end{array}
\]
Binary Search

• Assume we are searching for the value $x$ in the sorted array $data$
• We start by putting $left$ equal to 0, and $right$ equal to $data.length-1$
• Then we are sure that the element we are looking for is somewhere between $data[left]$ and $data[right]$
• Then we calculate the value $middle = (left+right)/2$ and look at the element $data[middle]$
• If $data[middle] \geq x$, then $x$ must lie between $data[left]$ and $data[middle]$
• If $data[middle] \leq x$ then $x$ must lie between $data[middle]$ and $data[right]$
• Therefore we have located which half of the array contains $x$
Example of binary search

$L=0$  
$L=7$  
$L=11$  
$L=11$  
$L=11$

$(L+R)/2=7$  
$(L+R)/2=11$  
$(L+R)/2=13$  
$R=15$  
$R=15$  
$R=15$  
$R=13$
public static boolean binarySearch(int[] data, int value) {
    int left = 0;
    int right = data.length-1;
    while (right - left > 1) {
        int middle = (left + right)/2;
        if (data[middle] >= value)
            right = middle;
        else
            left = middle;
    }
    if (data[left] == value || data[right]==value)
        return true;
    else
        return false;
}
Java implementation

int left = 0;
int right = data.length-1;

• This initializes the left and right indices so they are equal to the first index and the last index in the array
• If value is in the array anywhere, then it must somewhere “between” these locations
Java implementation

```java
while (right - left > 1) {
    int middle = (left + right)/2;
    if (data[middle] >= value)
        right = middle;
    else
        left = middle;
}
```

- As long as there is still some space between the indices, check the data item as close as possible to the middle of the portion of array where `value` must be located if it is in the array at all
- Then update `left` or `right` accordingly so that the portion of the array containing `value` has been “narrowed down”
Java implementation

```java
if (data[left] == value || data[right]==value)
    return true;
else
    return false;
```

Now that the left and right indices are next to each other, the element must either be equal to `data[left]` or `data[right]` in which case the method returns `true` or it is not actually in the array at all in which case it returns `false`.
Performance of binary search

• Each iteration reduces the effective range by half
• Therefore if the array has $n$ entries the algorithm will take at most $(1 + \log_2(n))$ iterations
• If the array has 1024 entries, then it will take at most 11 steps, as opposed to 1024 for the linear search
• If the array has 10,000,000 entries then the binary search will take at most 24 steps instead of 10,000,000!!
• Humans do an approximate binary search when using phone books, dictionaries and playing number guessing games
CITS 1200 Java Programming

Putting it together
Scope of this lecture

- A sample problem
- Specification
- Design
- Implementation
Problem

- A university lecturer wants to statistically analyse the marks obtained by students in an exam or test
- He asks his Java students to design a class that will permit him to
  - Enter marks from an exam or test
  - Calculate statistics such as number of marks, their average and standard deviation
  - Find the percentage pass rate for the test
  - Find the number of students who achieved each grade
Requirements

- The first step is to make the requirements precise by detailing exactly what the user needs
  - In particular you need to determine exactly what can be assumed and what the user needs to be able to alter or customize

- Sample questions to ask
  - Is the lecturer always able to enter all the marks at once, or will it be necessary to allow the user to add a few marks at a time?
  - Are the marks always out of 100?
  - Is the pass mark always 50%?
  - Can we assume that the UWA grading scheme (HD, D, CR, P etc) is the one to be used?
Refined specifications

- User wishes to analyse the marks for just one particular exam or test
- The user must be able to specify an entire collection of marks on construction, and also add additional marks later (for example, students sitting the test late)
- The maximum mark for the test must be a user-specified parameter, rather than a fixed value
- The pass mark is always 50% of the maximum mark
- The grading scheme can be assumed to be the one used by UWA
Design: classes

- Normally the first stage of an object-oriented design process is the identification of classes
  - Mapping the objects in the problem domain into Java classes
  - For large projects this is often a difficult and highly skilled task
- In this case we only have a tiny one-class program, so we need not spend much time on the design
- Objects of the class MarksAnalyser will store the marks for a single test or exam, and respond to all the statistical queries
Design: methods

• The next step is to decide on what the methods are - this should normally be done before starting to write any code
  – This means determining its signature - i.e. its return type, name and formal parameters

• Some of them are very straightforward:
  – public int numberMarks()
    • Returns the number of known marks
  – public double averageMarks()
    • Returns the average of all the known marks
  – public double sigmaMarks()
    • Returns the standard deviation of all the known marks
  – public double passRate()
    • Returns the percentage of students who passed the exam or test
Entering the marks

- How should the user enter the collection of marks?
  - On construction? After construction?
- What would be the “normal” usage of this class?
  - After the test, the lecturer will have a whole collection of marks to analyse, and then subsequently a few additional marks may come in from non-standard students
- So the solution that most closely mimics the expected use:
  - public MarksAnalyser(int[] marks)
    - The constructor should accept an initial array of marks
  - public void addMark(int mark)
    - And a separate method should permit additional marks
What about non-standard maximums?

- What if the test is out of 20 or 60 or something non-standard?
- The lecturer will know this on construction of the object and it will not change later, so this is a parameter that the user should specify on construction
  - public MarksAnalyser(int[] marks, int maxMark)
    - This constructor allows the client to specify the array of marks together with the maximum mark
    - This constructor is *additional* to the previous constructor, which will use a default maximum mark of 100.
The grades

- The grades present an interesting problem with an interesting solution
- UWA’s grade system is as follows:

<table>
<thead>
<tr>
<th>Grade Type</th>
<th>Grade</th>
<th>Percentage Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher Distinction</td>
<td>HD</td>
<td>80% - 100%</td>
</tr>
<tr>
<td>Distinction</td>
<td>D</td>
<td>70% - 79%</td>
</tr>
<tr>
<td>Credit</td>
<td>CR</td>
<td>60% - 69%</td>
</tr>
<tr>
<td>Pass</td>
<td>P</td>
<td>50% - 59%</td>
</tr>
<tr>
<td>Fail N+</td>
<td>N+</td>
<td>45% - 49%</td>
</tr>
<tr>
<td>Fail N</td>
<td>N</td>
<td>0% - 44%</td>
</tr>
</tbody>
</table>
What is the right signature?

We will return to this issue in a later lecture

- public int numberGetting(grade)

- How do we want the client to *specify* the grade when he asks for the number of students with, say, a distinction?

- We *could* use a *String* here. If the *MarksAnalyser* was called *ma*, then the client would specify
  
  ma.numberGetting(“D”);

- This solution works, but is not very elegant or robust
Implementation

• Once the design is completed, we can start with implementation
• This involves writing the actual code for
  – Instance variables that store the data
  – Constructor(s) to create the objects
  – The methods we have designed
Instance variables

- Instance variables should only be used to store persistent data relevant to the object
  - Persistent means data that must be preserved between method calls
  - The value of an instance variable should always be meaningful even when the object is not actually running any methods
- You should NOT use instance variables
  - As loop variables
  - As temporary variables used by a particular method
  - To communicate between methods
- In this case the persistent data is the marks and the maximum mark for the test
Storing the marks

- The first thing that comes to mind is to simply save all the marks in an array

```java
public MarksAnalyser {
    private int[] marks;
    private int maximumMark;
}
```
Storing the marks

• The constructor could then just copy the array of marks supplied by the client to the instance variable

```java
public MarksAnalyser {
    private int[] marks;
    private int maximumMark;

    public MarksAnalyser(int[] marks, int max) {
        this.marks = marks;
        maximumMark = max;
    }
}
```
But this causes problems…

- This intuitive idea turns out to be a bad idea because it is now very hard to add a mark to this collection.
- An array in Java cannot be resized and so adding a single mark would involve creating a new array that was one element longer and then adding the new mark to the end of the array.
- But in fact, we don’t even need or want to store all the marks, but simply their frequencies..i.e we want to know “how many 0s”, “how many 1s”, … , “how many 100s”
Store a frequency array instead

public MarksAnalyser {
    private int[] freq;
    private int maximumMark;
}

• Doing this means that the constructor will now have to create the frequency array and then populate it with the initial values; this is best done by using a private helper method
Initial setup

// Constructor
public MarksAnalyser(int[] marks, int max) {
    maximumMark = max;
    freq = new int[max + 1];
    enterInitialMarks(marks);
}

// Helper
private void enterInitialMarks(int[] marks) {
    for (int i=0; i<marks.length; i++) {
        freq[marks[i]]++;
    }
}
The helper

- Notice how the helper updates the frequency array with a loop containing just the single line
  \[ \text{freq}[\text{marks} [i]]++; \]
- How does this work? If \text{marks} is

| 37 | 82 | 96 | 45 | 66 | 59 |

then this loop has the same effect as

\[ \text{freq}[37]++; \text{freq}[82]++; \text{freq}[96]++; \]

etc.
Now updating is easy

• The update method is now trivial

    public void addMark(int mark) {
        freq[mark]++;
    }

Remember that \texttt{x++} is just shorthand notation for \texttt{x = x+1}
The default constructor

• The default constructor does everything that the first constructor does, but the maximum mark is set to be 100

    public MarksAnalyser(int[] marks) {
        maximumMark = 100;
        freq = new int[100 + 1];
        enterInitialMarks(marks);
    }

But this is just repeating code from the other constructor!
Can we just call the other constructor directly?
Another use of this

```java
public MarksAnalyser(int[] marks) {
    this(marks, 100);
}
```

• When the word this is used like a method name in a constructor it means to call one of the other constructors

• It is usually used exactly for this purpose - to provide a number of constructors that make sensible default choices for certain values, all of which use a single “most general” constructor
The remaining methods

• Once the correct structure has been set up, the remaining methods are fairly easy
• The total number of marks currently “stored” in the analyser is the sum of the frequencies.

```java
public int numberMarks() {
    int total = 0;
    for (int i=0; i<=maximumMark; i++){
        total = total + freq[i];
    }
    return total;
}
```
Averages logic

- The average is the total of all the marks divided by the number of marks,
- If we stored each mark individually, we would simply add them all up individually
- But as they are frequencies, we must remember that, for example, 7 students scoring 65 each means that we add $7 \times 65$ to the total
- Therefore $freq[i]$ students each scoring $i$ marks contributes $freq[i] \times i$ to the total
- We simply use the existing numberMarks() method to find the number of marks
Averages code

public double averageMark() {

    int totalMarks = 0;

    for (int i=0; i<=maximumMark; i++) {
        totalMarks = totalMarks + freq[i] * i;
    }

    return (double) totalMarks / this.numberMarks();
}

Maybe we should think about keeping an instance variable to keep track of the number of marks?
Pass Rate Logic

• The pass rate is the percentage of students who have passed the test
• The specifications say that 50% of the maximum mark is a passing mark
• So the basic idea is
  – Calculate the mark needed to pass
  – Count the number of students who got that mark or more
  – Divide by the total number of students
  – Multiply by 100 to get a percentage (rather than a proportion)
• What is the pass mark if the test is marked out of 15?
public double passRate() {

    int passMark;
    if (maximumMark % 2 == 0) {
        passMark = maximumMark / 2;
    } else {
        passMark = (maximumMark / 2) + 1;
    }

    int numPassed = 0;
    for (int i=passMark; i<=maximumMark; i++) {
        numPassed = numPassed + freq[i];
    }

    return 100d * numPassed / numberMarks();
}
Finding the pass mark

- The pass mark depends on whether the maximum mark is even or odd

```java
int passMark;

if (maximumMark % 2 == 0) {
    passMark = maximumMark / 2;
} else {
    passMark = (maximumMark / 2) + 1;
}

or

int passMark;

passMark = (maximumMark + 1) / 2;
```
Counting the number of passing students

A straightforward loop starting at the pass mark and ending at the maximum mark that counts the number of students that got each mark

```java
int numPassed = 0;

for (int i=passMark; i<=maximumMark; i++) {
    numPassed = numPassed + freq[i];
}
```
Calculate and return percentage

- Note the use of 100d to force the calculation to be done as a double rather than as an integer division

```java
return 100d * numPassed / numberMarks();
```
CITS 1200 Java Programming

Recursion I
Scope of this lecture

- Concept of recursion
- Simple recursive programs
- QuickSort
Recursion

- We have already seen that a method in a class can call other methods, either in the same or other classes.
- However a method in a class can also call itself.
- This self-referential behaviour is known as recursion.
- Recursion is an extremely powerful technique for expressing certain complex programming tasks, as it provides a very natural way to decompose problems.
- Despite this, there are costs associated with recursion and the careful programmer will always be aware of these.
The simplest example

- Consider the problem of computing the factorial function

\[ n! = n(n-1)(n-2)\ldots(1) \]

- So 2! = 2, 3! = 6, 4! = 24, 5! = 120, 6! = 720 and so on…
- However rather than define factorial in this way, we could give an equivalent, recursive definition

\[ 0! = 1 \]

\[ n! = n \times (n-1)! \]

- This definition is recursive because it uses factorial in the definition of factorial
public long factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n*factorial(n-1);
}

We use long just to give the method slightly greater range.

The method factorial calls itself, but with a smaller argument.
What happens in the method call?

\[
\text{factorial}(4) = 4 \times \text{factorial}(3)
\]

\[
\text{factorial}(3) = 3 \times \text{factorial}(2)
\]

\[
\text{factorial}(2) = 2 \times \text{factorial}(1)
\]

\[
\text{factorial}(1) = 1 \times \text{factorial}(0)
\]

\[
\text{factorial}(0) = 1
\]
Local variables

- Each recursive call to the method creates a “new copy” of the method – all the local variables are created again
- The compiler keeps track of them all so that the programmer cannot mix them up

\[
\text{factorial}(4) \text{ has a parameter } n \text{ equal to 4} \\
\text{factorial}(3) \text{ has its own parameter } n, \text{ equal to 3} \\
\text{factorial}(2) \text{ has its own parameter } n, \text{ equal to 2} \\
\text{factorial}(1) \text{ has its own parameter } n, \text{ equal to 1}
\]

- This means that you should not use recursion to replace straightforward iteration (like factorial), particularly if the loops are performed thousands or millions of times.
Ingredients for recursive definition

• Every valid recursive method definition needs two parts – the base case and the recursive part
• The base case gives the value that the method should return for some specified parameters
  – Usually the base case represents some sort of “trivial” case, such as a zero parameter or an empty list etc.
• The recursive part expresses the value to be returned in terms of another call to the same method, but with different parameters
  – To work correctly, the “different parameters” must be closer to the base case (in some sense)
Order is important

```java
public long factorial(int n) {
    return n * factorial(n - 1);

    if (n == 0) {
        return 1;
    }
}
```

You must put the base case before the recursive part of the definition or bad things will happen – in this case a stack overflow will result!
More than one base case

• There is no need to have just one base case
• The (mathematical) definition of the Fibonacci numbers is a recursive definition with two base cases

\[
F_0 = 1 \\
F_1 = 1 \\
F_n = F_{n-1} + F_{n-2}
\]

• This gives the sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, …
Be careful though

public int fibonacci(int n) {
   if (n == 0 || n == 1) {
      return 1;
   }
   else {
      return fibonacci(n-1)+fibonacci(n-2);
   }
}

This looks fine, but in fact is disastrously slow because it unnecessarily repeats some calculations over and over again!
Binary Search

- Binary search can be expressed recursively in a very natural fashion, because we repeatedly perform the same operation of calculating the middle element and then searching in an array of half the size.
- What is the “simplest case” in this situation?
- The length of the array is the parameter that is reduced at each stage of binary search and so the base case is when the left and the right bounds of the array are adjacent.
- In this situation it is trivial to determine whether or not the element we are looking for is in the array or not.
boolean binarySearch(int[] a, int lf, int rt, int val) {
    if (rt - lf == 1)
        return a[lf] == val || a[rt] == val;

    int mid = (lf+rt)/2;
    if (a[mid] > val)
        return binarySearch(a,lf,mid,val);
    else
        return binarySearch(a,mid,rt,val);
}

Binary Search
Public interface

- This method would usually be made **private**
- The public interface would be the simpler

```java
public boolean binarySearch(int[] a, int val)
```

- Its only role would be to call the recursive version

```java
public boolean binarySearch(int[] a, int val) {
    return binarySearch(a, 0, a.length-1, val);
}
```
A recursive sorting algorithm

- Suppose you had to sort the following array with 16 elements

  53 49 57 35 18 11 23 62 71 90 95 87 77 92 83

- Just before starting, you notice that the array has a very special structure

  53 49 57 35 18 11 23 62 71 90 95 87 77 92 83

  All smaller than 62
  All larger than 62
Divide and conquer

- We can now divide the problem into two smaller problems

53 49 57 35 18 11 23

Sort

11 18 23 35 49 53 57

The fence

62

Sort

11 18 23 35 49 53 57 62

71 90 95 87 77 92 83

Sort

71 77 83 87 90 92 95

- The two “half-size” problems take much less than half the time!
But what if the array is not in this nice form?

• Then we put it into this nice form!

• First we choose an element to be the “fence”, and then we adjust the array so that the fence is in the correct position, everything to the left of the fence is smaller than it, and everything to the right of the fence is larger than it

Choose the fence (we explain how later)
Find out-of-place elements

Now increase L until reaching an element *bigger* than the fence

And decrease R until reaching an element *smaller* than the fence
Then swap them over

L
R

62 49 24 18 98 53 90 11 57 95 82 87 35 92 77 71

Move

L
R

62 49 24 18 98 53 90 11 57 95 82 87 35 92 77 71

Exchange

L
R

62 49 24 18 35 53 90 11 57 95 82 87 98 92 77 71
Repeat

Move

```
62 49 24 18 35 53 90 11 57 95 82 87 98 92 77 71
```

Exchange

```
62 49 24 18 35 53 57 11 90 95 82 87 98 92 77 71
```

Until \( R < L \)

```
When $R < L$

When $R < L$, then every element to the right of $L$ is larger than the fence, and every element (except the fence) to the left of $L$ is less than the fence.

Swap the fence and element $R$.
public void partition(int[] a) {
    int fence = a[0];
    int left = 1;
    int right = a.length-1;
    while (right >= left) {
        while (left <= right && a[left] <= fence)
            left++;
        while (right >= left && a[right] >= fence)
            right--;
        if (right > left) {
            int swap = a[left];
            a[left] = a[right];
            a[right] = swap;
        }
    }
    a[0] = a[right];
    a[right] = fence;
}
public void partition(int[] a) {
    int fence = a[0];
    int left = 1;
    int right = a.length-1;

    while (right >= left) {
        while (left <= right && a[left] <= fence)
            left++;
        while (right >= left && a[right] >= fence)
            right--;

        if (right > left) {
            int swap = a[left];
            a[left] = a[right];
            a[right] = swap;
        }
    }

    a[0] = a[right];
    a[right] = fence;
}
public void partition(int[] a) {
    int fence = a[0];
    int left = 1;
    int right = a.length-1;
    while (right >= left) {
        while (left <= right && a[left] <= fence)
            left++;
        while (right >= left && a[right] >= fence)
            right--;
        if (right > left) {
            int swap = a[left];
            a[left] = a[right];
            a[right] = swap;
        }
    }
    a[0] = a[right];
    a[right] = fence;
}
QuickSort

• **QuickSort** is a recursive sorting method defined as follows:
  
  • To sort an array,
    – Partition the array around some fence
    – **QuickSort** the elements of array *before* the fence position
    – **QuickSort** the elements of array *after* the fence position

• This is a recursive definition because we have *used* **QuickSort** in the *definition* of **QuickSort**

• We have only given the recursive part of the definition; what is the base case?
  – This is easy – if the array has 0 or 1 elements to be sorted, then we do not need to do anything!
In-place sorting

private static int partition(int[] a, int start, int finish) {
    int fence = a[start];
    int left = start + 1;
    int right = finish;

    // omitted code is identical to before

    a[start] = a[right];
    a[right] = fence;

    return right;
}
private static void quickSort(int[] a, int start, int finish) {
    if (finish-start > 0) {
        int fence_position = partition(a,start,finish);
        quickSort(a,start,fence_position-1);
        quickSort(a,fence_position+1,finish);
    }
}

public static void quickSort(int[] a) {
    quickSort(a,0,a.length-1);
}
The call `quickSort(a)`

First calls `partition(a,0,15)`, which changes `a` to

```
11 49 24 18 35 53 57 62 90 95 82 87 98 92 77 71
```

and returns 7 as the fence position

- `quickSort(a,0,6)` sorts the left half
- `quickSort(a,8,15)` sorts the right half

```
11 49 24 18 35 53 57 62 90 95 82 87 98 92 77 71
```
Timing comparison

- The **QuickSort** method is enormously faster than **InsertionSort** or **BubbleSort** – in fact so fast, that the inaccuracy of `System.currentTimeMillis()` makes it impossible to measure the time to sort just one array of these small sizes.
Comments on QuickSort

• Works well when the fence position ends up being roughly half way through the array, as the two sub-problems are then about equal in size

• Our choice of fence – position 0 – means that the worst case for QuickSort is when the array is already ordered

• In practice, it pays to avoid this case
  – Pick the fence randomly
  – Randomly permute the array before sorting with QuickSort
  – Sample a small portion of the array, and choose the median as fence

• Average number of comparisons is \( n \log_2 n \)
CITS 1200 Java Programming

Recursion II
Scope of this lecture

- Recursive Listing
- Recursively defined classes

References:
- Barnes, *Object-Oriented Programming with Java*, Section 10.10
Greatest Common Divisor

• One of the most famous algorithms ever – Euclid’s algorithm
• The greatest common divisor of two integers is the largest number that exactly divides both of them
• Examples:
  – GCD of 8 and 12 is 4
  – GCD of 5 and 20 is 5
  – GCD of 7 and 11 is 1
  – GCD of 30 and 110 is 10
Recursive formulation

• Given two numbers $a$ and $b$ (with $a > b$) we can find the *quotient* and *remainder* when dividing $a$ by $b$

$$a = q \cdot b + r$$

• If $r = 0$, then $b$ exactly divides $a$, and so $\text{gcd}(a, b) = b$

• If $r$ is not zero, then the greatest common divisor of $a$ and $b$ is equal to the greatest common divisor of $b$ and $r$

$$\text{gcd}(a, b) = \text{gcd}(b, r)$$

• A smaller version of the same problem – *recursion*!
Euclid’s Algorithm in Action

- What is $\gcd(110, 30)$?
- Find quotient and remainder when dividing 110 by 30
  \[ 110 = 3 \cdot 30 + 20 \]
- What is $\gcd(30, 20)$?
- Find quotient and remainder when dividing 30 by 20
  \[ 30 = 1 \cdot 20 + 10 \]
- What is $\gcd(20, 10)$?
- Find quotient and remainder when dividing 20 by 10
  \[ 20 = 2 \cdot 10 + 0 \]
- Remainder is zero, so finished! The $\gcd$ is 10
Euclid’s Algorithm in Java

```java
int gcd(int a, int b) {
    if (b == 0)
        return a;

    int r = a % b;
    return gcd(b, r);
}
```
Not robust at the moment

- Before being “released”, such code should be made more robust
  - Should reject negative arguments by throwing an IllegalArgumentException
  - Can deal safely with $a < b$, just by swapping around $a$ and $b$

```java
public int gcd(int a, int b) {
    if (a < 0 || b < 0)
        throw new IllegalArgumentException();

    if (b < a)
        return gcd2(a,b);
    else
        return gcd2(b,a);
}
```

Use `gcd2` for the method of the previous slide, and use `gcd` as the “gatekeeper”
Recursively defined data structures!

```java
public class Employee {
    private int tax_file_number;
    private Employee supervisor;
    private Employee[] staff;
}
```

Every employee has a supervisor, who is an Employee and staff they directly manage, who are themselves Employees!
An organisational chart

- Alf
  - Betty
    - Ethel
      - Jack
    - Fred
      - Keith
  - Charles
    - George
  - Diane
    - Hilary
    - Imogen
      - Lucy
        - Mark
        - Nora
Employee objects

supervisor = null
staff[0] = Betty
staff[1] = Charles
staff[2] = Diane

supervisor = Alf
staff[0] = Ethel
staff[1] = Fred

Alf

Betty
Rank of Employees

• Consider the following method

```java
public String title() {
    if (supervisor == null)
        return "President";
    else
        return "Vice-"+supervisor.title();
}
```
Cascading method calls

Alf.title() returns "President"

Betty.title() returns
   "Vice-"+Alf.title() which is "Vice-President"

Mark.title() returns
   "Vice-"+Lucy.title() which is
   "Vice-"+"Vice-"+Hilary.title() which is
   "Vice-"+"Vice-"+"Vice-"+Diane.title() which is
   "Vice-"+"Vice-"+"Vice-"+"Vice-"+Alf.title()
which is "Vice-Vice-Vice-Vice-President"
Pass the buck

• In this situation most method calls are achieved by simply “passing the buck” – asking another object to perform some calculation

• Consider the problem of Alf trying to find out how many subordinates he has (staff, staff-of-staff, staff-of-staff-of-staff etc)

• The “pass the buck” method is to
  – Ask Betty how many subordinates she has
  – Ask Charles how many subordinates he has
  – Ask Diane how many subordinates he has
  – Add those numbers together and add 3 (for Betty/Charles/Diane)
The buck stops here

• Of course, in order to get an answer eventually it is necessary that there be a “base-case” – someone must just answer the question without passing it on to someone else

• Who can definitively answer the question about how many subordinates they have without having to refer it to someone else?
  – Ethel has no staff, so can immediately answer “0”
  – Similarly for Jack, Keith etc and all of the other non-managers

• The buck stops at the bottom of the organisation!
public int subordinateCount() {
    if (staff == null)
        return 0;
    else {
        int num = 0;
        for (int i=0; i<staff.length; i++) {
            num = num + 1 + staff[i].subordinateCount();
        }
        return num;
    }
}
Listing Things

• Recursion is particularly useful for tasks that involve trying “all possible combinations” or listing things “in all possible ways”

• As an example, we will consider the problem of finding all the ways of partitioning an integer into smaller parts

• A partition of an integer $n$ is a collection of positive integers that sum up to $n$

1+1+1+1
1+1+2
1+3
2+2
4

There are 5 different partitions of the number 4 (we usually assume the numbers are listed in increasing order)
Listing Partitions

• Suppose we want to write a method to list all the partitions of an integer \( n \)

```java
public void listPartitions(int n)
```

• How could we do this?
• We can easily find all the partitions with two parts, using a loop

```java
for (int i=1; i<=n; i++) {
    if (n-i >= i)
        System.out.println(i + " + " + (n-i));
}
```

1+1
1+3
2+2
Java Programming

Three parts
• To do three parts, we need a nested loop
public void list3Parts(int n) {
for (int i=1;i<=n;i++)
for (int j=i;j<=n;j++)
if (n-i-j>=j)
System.out.println(i+"+"+j+"+"+(n-i-j));
}

The output from
list3Parts(8)

© CSSE, Gordon Royle 2008

18


When does it end?

- Clearly we could also write `list4Parts` using 3 nested loops, `list5Parts` using 4 nested loops etc but this is no solution
- We want the program to work for any integer n, so the number of parts cannot be fixed in advance
- Recursion provides an elegant solution to this problem
- The key observation is that if
  \[ p_1 + p_2 + \ldots + p_k \]
  is a partition of n, then
  \[ p_2 + \ldots + p_k \]
  is a partition of \( n - p_1 \)
Recursive solution

• This simple observation can be turned into a recursive algorithm
• To list all the partitions of $n$
  – Choose $p_1 = 1$ and find all the partitions of $n-1$
  – Choose $p_1 = 2$ and find all the partitions of $n-2$ (with smallest part 2)
  – Choose $p_1 = 3$ and find all the partitions of $n-3$ (with smallest part 3)
  – etc
• For example, consider the partitions of 4 that have $p_1 = 1$

1+1+1+1
1+1+2
1+3

These are the partitions of 3
public static void partition(int[] parts, int num, int n) {
    if (n == 0) {
        for (int i=0; i<num; i++)
            System.out.print(parts[i]+" ");
        System.out.println();
    } else {
        int min = (num == 0) ? 1 : parts[num-1];
        for (int p1=min; p1<=n; p1++) {
            parts[num] = p1;
            partition(parts, num+1,n-p1);
        }
    }
}
In Java

```java
public void partition(int[] parts, int num, int n)
```

- The array `parts` will contain the list of parts
- `n` is the (rest of) the number to partition
- `num` is the number of parts assigned so far
In Java

public static void partition(int[] parts, int num, int n) {
    if (n == 0) {
        for (int i=0; i<num; i++)
            System.out.print(parts[i]+" ");
        System.out.println();
    } else {
        int min = (num == 0) ? 1 : parts[num-1];
        for (int p1=min; p1<=n; p1++) {
            parts[num] = p1;
            partition(parts, num+1, n-p1);
        }
    }
}

Base case

Recursive part
In Java

```java
public static void partition(int[] parts, int num, int n) {
    if (n == 0) {
        for (int i=0; i<num; i++)
            System.out.print(parts[i]+" ");
        System.out.println();
    } else {

    }
}
```

• The base case is when n=0 – there is nothing more to partition; all that needs to be done is to print out the partition that has been constructed which is contained in the array

• The method will end, and thus return to the calling method
In Java

• The recursive part first finds the smallest part that it is allowed to use, with the ternary operator `?:`

• Then a loop chooses all the possible values for `p1`, and calls `partition` again, but with the smaller number `n-p1` left to partition.

```java
int min = (num == 0) ? 1 : parts[num-1];
for (int p1=min; p1<=n; p1++) {
    parts[num] = p1;
    partition(parts,num+1,n-p1);
}
```
The ternary operator `?:`

- This operator has the form
  `<boolean_expression> ? <expr1> : <expr2>`
- It has the value `<expr1>` if the `<boolean_expression>` is true, and `<expr2>` otherwise
- Therefore
  `(num == 0) ? 1 : parts[num-1]`
  has the value 1 (if `num` is equal to 0) or `parts[num-1]` otherwise
  - This is the smallest value that is allowed to be used as a part in the rest of the partition
Initial call

- This method would be called by a public method that hides the details of the implementation

```java
public void listPartitions(int n) {
    int[] parts = new int[n];
    partition(parts, 0, n);
}
```
Tracing a recursive call

listPartition(4)
    partition({0,0,0,0},0,4)
        partition({1,0,0,0},1,3)
            partition({1,1,0,0},2,2)
                partition({1,1,1,0},3,1)
                    partition({1,1,1,1},4,0) -> output 1+1+1+1
                        returns
                        returns
            returns
        returns
    returns
    partition({1,1,2},3,0) -> output 1+1+2
        returns
    returns
    partition({1,2},2,1)
        returns
    returns
    partition({1,3},2,0) -> output 1+3
        returns
    returns
<and so on>
Tracing a recursive call

listPartition(4)

  partition({0,0,0,0},0,4)

    partition({1,0,0,0},1,3)

      partition({1,1,0,0},2,2)

        partition({1,1,1,0},3,1)

          partition({1,1,1,1},4,0) -> output 1+1+1+1

          returns

          returns

        partition({1,1,2,0},3,0) -> output 1+1+2

        returns

      returns

    partition({1,2,0,0},2,1)

    returns

    partition({1,3,0,0},2,0) -> output 1+3

    returns

  returns

<and so on>
Partitions of 10