Abstract Data Types

This lecture will

- Show how an abstract data type can be implemented as a Java class;
- Show how one class can be used by another;
- Introduce the **stack** data structure;
- Introduce the concept of a **wrapper class**, and the related concepts of **autoboxing** and **unboxing**.
- Look at a complicated design example

A complex number class

- Complex numbers often arise in mathematics. They have the form $a + ib$, where $i = \sqrt{-1}$, a is called the real part and b is called the imaginary part.
- For a complex number class we need to identify its variables and methods
- The variables are:
  - **`realPart`**
  - **`imagPart`**

Methods of the complex number class

- **`getReal()`** and **`getImag()`**, the accessors
- **`copy()`** and **`toString()`**
- **`add()`**, **`subtract()`**, **`multiply()`**, **`divide()`** with their obvious meanings
- **`Conj()`**, **`abs()`**, **`angle()`** – more obscure values of complex numbers for completeness

Don't panic if you didn't do A Level maths. The class matters not the maths

The `Complex` class – constructor

```
public class Complex {

   private double realPart;
   private double imagPart;

   /**
   * Creates an instance of the Complex class with
    * specified  values
   * @param   r      double       real part
   * @param   i      double       imaginary part
   */
   public Complex(double r, double i) {
      realPart = r;
      imagPart = i;
   }
```

The constructor has two double parameters. In this case the user can be expected to know which order they come in because complex numbers are always written real then imaginary but it can be a problem in other cases

The Complex class – get methods

```
/**
* returns the real part of the complex number
* @return double the real part of the complex number
*/
public double getReal() {
   return realPart;
}
/**
* returns the imaginary part of the complex number
* @return double the imaginary part of the complex
*                 number
*/
public double getImag() {
   return imagPart;
}
```

The Complex class – add

```
/**
* returns the sum of the complex number and another
* complex number
* @param c    Complex  the complex number to add
* @return Complex     the sum of the complex numbers
*/
public Complex add(Complex c) {
   return new Complex(
       realPart+c.getReal(),
       imagPart+c.getImag());
}
```

The Complex class – subtract

```
/**
* returns the difference between the complex number
* and another complex numbers
* @param c Complex  the complex number to subtract
* @return Complex   the difference of the complex
*                   numbers
*/
public Complex subtract(Complex c) {
   return new Complex(
       realPart-c.getReal(),
       imagPart-c.getImag());
}
```

The Complex class – multiply

```
/**
* works out the product of the complex number and
* another complex number
* @param c Complex  the complex number to multiply by
* @return Complex   the product of the complex numbers
*/
public Complex multipliedBy(Complex c) {
   return new Complex(
       realPart*c.getReal()-imagPart*c.getImag(),
       realPart*c.getImag()+imagPart*c.getReal());
}
```

The `Complex` class – divide

```
/**
* Works out the the complex number divided by another
* complex number
* @param c Complex  the complex number to divide by
* @return Complex   the division of two complex numbers
*/
public Complex dividedBy(Complex c) {

   double d = c.getReal()*c.getReal() +
              c.getImag()*c.getImag();

   return new Complex(
      (realPart*c.getReal()+imagPart*c.getImag())/d,
      (realPart*c.getReal()-realPart*c.getImag())/d);
}
```

The `Complex` class – `copy` and `toString`

```
/**
* returns a copy of a complex number
* @return Complex a copy of a complex number
*/
public Complex copy() {
   return new Complex(realPart,imagPart);
}
/**
* converts a complex number to a string for display
* @return String          a string representing the
*                         complex number
*/
public String toString() {
   if (  imagPart < 0.0  )
      return realPart+(imagPart+"i");
   else
      return realPart+"+"+imagPart+"i";
}
```

Writing a test harness

- We provide a **`main`** method that reads a pair of complex numbers and tests each method of the **`Complex`** class
- Note that **`Complex`** is not intended to be invoked directly by the Java interpreter (rather, instances of it will be created by other classes) – but by providing a **`main`** method we can now do so
- This allows us to test the class in isolation before it is integrated into a larger system

The `Complex` class – test harness

```
public static void main(String args[]) {
   EasyReader keyboard = new EasyReader();
   do {

      // create two complex numbers
      // display the two numbers
      // test the accessor methods
      // test the operations on the numbers

   } while (keyboard.readBoolean("Another go?: "));

}
```

Running the test harness (input in yellow)

```
> java Complex
First number:
Enter real part: 3
Enter imaginary part: -2
Second number:
Enter real part: 6
Enter imaginary part: 9

c1 = 3.0-2.0i
c2 = 6.0+9.0i
real part of c1 = 3.0
real part of c2 = 6.0
imaginary part of c1 = -2.0
imaginary part of c2 = 9.0

c1+c2 = 9.0+7.0i
c1-c2 = -3.0-11.0i
c1*c2 = 36.0+15.0i
c1/c2 = 0.0-0.07692307692307693i
abs(c1) = 3.605551275463989
abs(c2) = 10.816653826391969
conj(c1) = 3.0+2.0i
conj(c2) = 6.0-9.0i
angle(c1) = -0.5880026035475675
angle(c2) = 0.982793723247329


Another go?: n
```

Implementation issues

- Note the way that mathematical expressions are formed:

```
Complex c1 = new Complex(2.4,1.3);
Complex c2 = new Complex(1.7,4.6);
Complex sum = c1.add(c2);
```

- We could also declare mathematical operations as class methods, rather than instance methods

Implementation issues

- The **static** version

```
public static Complex add(Complex c1,Complex c2){
    return new Complex(c1.getReal()+c2.getReal(),
                       c1.getImag()+c2.getImag());
}
```

- Now we can write expressions like this:

```
Complex sum = Complex.add(c1,c2);
```

Immutable classes

- The complex number class has accessor methods but no mutators
- It is an **immutable** class. Objects, once created, cannot be changed
- If you need a complex number with a different imaginary or real part you must create a new one
- Immutable classes are common e.g. **String** and **LocalDate**

A stack class

- Stacks are useful in many computer science applications, such as writing compilers.
- Java actually provides a **`Stack`** class in the collections framework – however, it is instructive to see how to implement a stack for ourselves.
- A stack is characterised by the property that only the top element on the stack is accessible.

Image from Wikipedia

The Stack's methods

- A **`Stack`** needs the following methods
  - **`isFull()`** returns true if the stack is full
  - **`isEmpty()`** returns true if the stack is empty
  - **`pop()`** removes an element from the top of the stack
  - **`push()`** inserts an element on the top of the stack
  - **`retrieve()`** returns a copy of the element on the top of the stack, without removing it.

The `Stack` class – constructor etc.

```
public class Stack {
   //Constant - the maximum size of the Stack
   private static final int MAX_ITEM = 10;

   // instance variables
   private int numElements;
   private Object[] items;

   /*
   * Constructor
   * @returns a new, empty stack
   */
   public Stack() {
      numElements = 0;
      items = new Object[MAX_ITEM];
   }
```

See later

The `Stack` class – `isFull` & `isEmpty`

```
/**
* Determines whether the Stack is full
* @return boolean   true if the Stack is full
*/
public boolean isFull() {
   return numElements==MAX_ITEM;
}

/**
* Determines whether the Stack is empty
* @return boolean true if the Stack is empty
*/
public boolean isEmpty() {
   return numElements==0;
}
```

The Stack class – the pop method

```
/**
* Removes the element from the top
* Stops with an error if the stack is empty
* @return Object the element on the top of the
*                          Stack
*/
public Object pop() {
   if (  isEmpty()  ) System.exit(0);
   numElements-=1;
   return items[numElements];
}
```

The Stack class – the push method

```
/**
* Puts an element on the top of the Stack
* The method stops with an error if the stack is
* full
* @param obj Object The thing to be added to the
*                         stack
*/
public void push(Object obj) {
   if (  isFull()  )  System.exit(0);
   items[numElements] = obj;
   numElements+=1;
}
```

The Stack class – retrieve method

```
/**
* Returns a reference to the item on the top of
* the Stack
* The contents of the Stack are not changed
* @return Object the item on the top of the Stack
*/
public Object retrieve() {
   if (  isEmpty()  ) System.exit(0);
   return items[numElements-1];
}
```

Test harness for the Stack class

```
public static void main(String args[]) {
   // variable declarations
   EasyReader keyboard = new EasyReader();
   Stack myStack = new Stack();

   // read five integers and push each onto the stack
   for (int i=0; i<5; i++) {
     int num=
         keyboard.readInt("Enter number "+(i+1)+": ");
     myStack.push(num);
   }

   // check the use of isEmpty and pop
   while (  !  myStack.isEmpty()  )
       System.out.println(  myStack.pop()  );
}
```

Output of the test harness

- Typical run of test harness shown on the right, with user input in yellow.
- Note that a stack is a **last-in first-out** (LIFO) data structure.
- Pushing a sequence of numbers onto the stack and then popping them off reverses their order.

```
>java Stack
Enter number 1: 5
Enter number 2: 4
Enter number 3: 3
Enter number 4: 2
Enter number 5: 1
1
2
3
4
5
```

Implementation issues

- We defined the stack as an array of type **`Object`**:

```
private Object[] items;
```

- All classes in Java are subclasses of the **`Object`** class; in other words, we can treat classes like **`String`** as “a kind of” **`Object`**.
- This is the concept of **inheritance** – see next term
- Using **`Object`**s makes the class more general; we can put anything on it

A puzzle

- In the test harness for the **`Stack`** we wrote

```
myStack.push(num);
```

but the signature for the push method is

```
public void push(Object obj)
```

and **`num`** had the type **`int`** which is not an object and yet it compiles

- The solution lies in **wrapper classes**, which act as object wrappers around primitive types

Wrapper types

- For every primitive type there is a corresponding **wrapper class** which represents a single value of that type
- We could write

```
Integer objectNum = new Integer(num);
```

or

```
Integer fortyTwo = new Integer(42);
```

Wrapper types and autoboxing

- To get an **`int`** back from an **`Integer`** object we can write

```
int numAgain = objectNum.intValue();
```

An instance method of **`Integer`**

- But it is unnecessary, conversion of primitive types to the wrapper type is done automatically; this is called **autoboxing**
- Wrapper types are also **unboxed** as required

```
int numAgain = objectNum;
objectNum = 97;
```

Wrapper classes

- All the basic types have wrapper classes

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

- You have been using their static methods

Character static methods

```
public static char toUpperCase(char ch)
public static char toLowerCase(char ch)

public static boolean isLowerCase(char ch)
public static boolean isUpperCase(char ch)

public static boolean isLetter(char ch)
public static boolean isDigit(char ch)
```

```
char c = keyboard.readChar();
if (  Character.isDigit(c)  )…
```

Object-oriented design – a case study

- We will develop a Java program to play the Game of Life. A cellular automaton devised by Conway in 1970. See www.wikipedia.org for more information
- The Game of Life takes place on a rectangular grid where cells are either empty or “alive”
- Simple rules are used to change the state of each cell in the grid over time

The Game of Life - rules

Each cell, which is either empty or alive, can be thought of as the centre of a 3 by 3 square grid of cells, which contains its eight neighbours

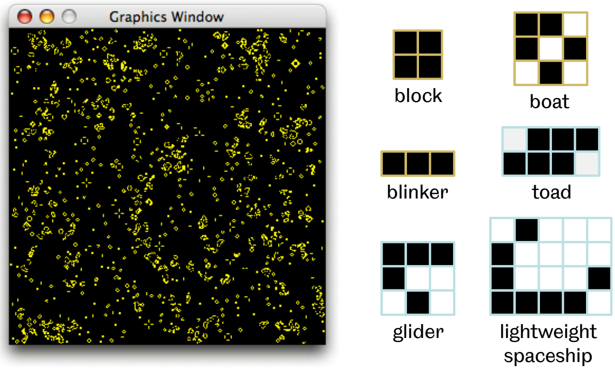
i. An empty cell at time ***t*** becomes alive at time ***t+1*** if and only if three neighbouring cells were alive at time ***t***

ii. A cell that is alive at time ***t*** remains alive at time ***t+1*** if and only if either two or three neighbouring cells were alive at time ***t***

The game of life - implementation

- For simplicity, the borders of the grid are assumed never to contain any live cells
- A simulation starts with a random configuration of cells in the grid. At each time step, a new state for the grid is created from the previous state by applying the rules and displayed
- After several generations, interesting behaviour occurs such as the appearance of repeating life forms

Output of the program



Object-oriented design – a case study

- We will use a simple OOD methodology based on Booch (1983). More advanced object-oriented design methodologies and notations will be covered in other modules.
- Go through the following steps:
  - Step 1: define the problem
  - Step 2: develop an informal strategy for solving the problem
  - Step 3: formalise the strategy

Step 2: develop an informal strategy

- Write an informal description of the problem solution using terminology from the problem space.
- In this case, our strategy is quite close to the original problem description.

The game of life takes place on a grid of pixels. At the start of the game, we initialise the grid by randomly setting each cell to be alive or dead. A new generation is then computed from the initial state by applying some simple rules. For each cell (but not cells at the edge of the grid), we determine how many of its neighbours are alive. We then use this information to set the value of the cell in the new state of the grid. The new state of the grid is displayed, and the old state is set to the new state. This process continues for the desired number of generations.

Step 3: formalise the strategy

- List the objects in the informal strategy. These correspond to **nouns**.
- List the methods (operations performed on objects) in the informal strategy. These correspond to **verbs**.
- Group together objects and methods into classes.
- This is usually an iterative process.

The problem description's nouns

The **game of life** takes place on a **grid of pixels**. At the start of the game, we initialise the grid by randomly setting each cell to be alive or dead. A new **generation** is then computed from the initial **state** by applying some simple rules. For each **cell** (but not cells at the edge of the grid), we determine how many of its neighbours are alive. We then use this information to set the value of the cell in the new state of the grid. The new state of the grid is displayed, and the old state is set to the new state. This process continues for the desired number of **generations**.

Possible classes

- Possible classes: game of life, grid, generation, state, cell
- Discount inappropriate classes

game of life | grid | ~~generation~~

~~state~~ | cell ?

The problem description's verbs

The *game of life* takes place on a *grid of pixels*. At the start of the game, we **initialise the grid by randomly setting each cell** to be alive or dead. A new *generation* is then **computed** from the initial *state* by **applying some simple rules**. For each *cell* (but not cells at the edge of the grid), we **determine how many of its neighbours are alive**. We then use this information to **set the value of the cell** in the new state of the grid. The new state of the grid is **displayed**, and the **old state is set to the new state**. This process continues for the desired number of *generations*.

Possible methods

- Possible methods: initialise, compute new generation, apply rules, determine how many neighbours are alive, set the value of a cell, display the grid, set the old state to the new state

initialise

compute new generation

apply rules

determine how many neighbours are alive

set the value of a cell

display the grid

set the old state to the new state

Object-oriented design solution

game of life

grid

cell

initialise

compute new generation

apply rules

determine how many neighbours are alive

set the value of a cell

display the grid

set the old state to the new state

Object-oriented design solution

game of life	grid	cell
initialise	initialise	initialise
compute new generation	display the grid	determine how many neighbours are alive
set the old state to the new state	apply rules	set the value of a cell
Update state		

Writing classes with method stubs

- Start by writing the classes with instance variables and class constants, but **stubs** for the methods (empty method body).
- At this stage it emerges that it is necessary to have a **`display`** method in the **`GameOfLife`** class which will display the current state of the game; we expect this just to call a method that displays the grid.

`GameOfLife` with method stubs

```
public class GameOfLife {
   private static final int MAX_GENERATIONS = 200;
   private static final int GRID_SIZE = 500;

   private Grid state;     // current grid state

   // constructor
   public GameOfLife() {}
   // initialise the game
   public void initialise() {}
   // update to the next generation
   public void updateState() {}
   // display the state of the grid
   public void display() {}
   // main method
   public static void main(String[] args) {}
}
```

The Grid - choice of data structure

- The grid should be represented as a 2D array of cells which will contain only boolean values (since each cell has a binary value, alive/dead)
- Do we really need a cell class?
- To keep things general the constructor for the grid (and therefore the game) will allow the size of the grid to be specified

`Grid` with method stubs

```
public class Grid {
   // the grid
   private boolean grid[][];

   // constructor, s is the size of the grid
   public Grid(int s) { grid = new boolean[s][s];}

   // set the cell at (i,j) to value b
   public void setCell(int i, int j, boolean b) { }
   // get the value of cell (i,j)
   public boolean getCell(int i, int j) { }
   // initialise the grid with random values
   public void initialise() { }
   // get the number of alive neighbours of (i,j)
   public int aliveNeighbours(int i, int j) { }
   // display
   public void display() { }
}
```


Developing algorithms for methods

- Use top-down design to develop algorithms for methods.
- Algorithm for initialising the grid:

1. *Set the border cells to be dead*
2. *for each row i*
3. *for each column j*
4. *set the element at (i,j) to alive/dead with equal probability*

The initialise method

```
public void initialise() {
  int gridSize = grid.length;
  //set the first and last row to be dead
  for  (int j=0; j<gridSize; j++)  {
    grid[0][j] = false;
    grid[gridSize-1][j] = false;
  }
  //the other rows should start and end with dead
  //but the cells in between have a 50% chance of
  //life
  for (int i=1; i<gridSize-1; i++) {
    grid[i][0] = false;
    grid[i][gridSize-1] = false;
    for (int j=1; j<gridSize-1; j++)
        grid[i][j]=Math.random()<0.5;
   }
}
```

The display method

```
public void display(EasyGraphics g) {
   for (int i=0; i<grid.length; i++)
     for (int j=0; j<grid.length; j++) {
         if (  grid[i][j]  )
             g.setColor(255,255,0);
         else
             g.setColor(0,0,0);
         g.plot(i,j);
      }
}
```

Finding the number of alive neighbours

- Algorithm:

1. *sum the number of alive cells in a 3x3 grid centred on (i,j)*
2. *if the element in the centre of the 3x3 grid is alive then subtract one from the sum*

- Step 1 refinement:

1.1 *set the sum to zero*
1.2 *for each row r between i-1 and i+1*
1.3 *for each column c between j-1 and j+1*
1.4 *if the element at (r,c) is alive then*
1.5 *add one to the sum*

The aliveNeighbours method

```
public int aliveNeighbours(int i, int j) {
   int sum=0;
   for (int r=-1; r<=1; r++)
     for (int c=-1; c<=1; c++)
       if (grid[i+r][j+c]) sum++;
   if (grid[i][j]) sum--;
   return sum;
}
```

- Writing **get** and **set** methods is easy.
- We also provide a test harness (**main** method) which enables us to test the **Grid** class independently.

Completed Grid class

```
import sheffield.*;
public class Grid {

   private boolean grid[][];

   public Grid(int gridSize) {
      grid = new boolean[gridSize][gridSize];
   }

   public boolean getCell(int i, int j) {…}
   public void setCell(int i, int j, boolean b) {…}
   public int getGridSize() {…}
   public void initialise() {…}
   public int aliveNeighbours(int i, int j) {…}
   public void display(EasyGraphics g) {…}

}
```

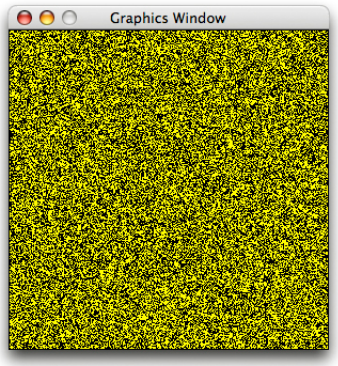
The Grid classes test harness

```
public static void main(String[] args) {
   final int GRID_SIZE = 300;
   EasyGraphics g = new

   EasyGraphics(GRID_SIZE,GRID_SIZE);
      Grid grid = new Grid(GRID_SIZE);
      grid.initialise();
      grid.display(g);
   }
}
```

```
>javac Grid.java
>java Grid
```

Output of the Grid test harness



The `GameOfLife` class

- Top-level algorithm:

1. *initialise the grid to have random contents*
2. *for each generation*
3. *display the grid in the graphics screen*
4. *update the grid using the rules*

- This gives us the **`main`** method:

```
public static void main(String[] args) {
   GameOfLife game = new GameOfLife(GRID_SIZE);
   game.initialise();
   for (int i=0; i<MAX_GENERATIONS; i++) {
      game.display();
      game.updateState();
   }
}
```

The `GameOfLife` constructor

- The constructor is straightforward; we just need an old and new grid state, and a graphics window:

```
public class GameOfLife {
   private static final int MAX_GENERATIONS = 200;
   private static final int GRID_SIZE = 500;

   private Grid state;
   private EasyGraphics graphics;

   public GameOfLife(int size) {
      state = new Grid(size);
      graphics = new EasyGraphics(size, size);
   }
   ……
```

Initialising & displaying the game state

- To initialise the game we set the grid to an initial random state:

```
public void initialise() {
   state.initialise();
}
```

- To display the game state, we just call the display method of the grid:

```
public void display() {
   state.display(graphics);
}
```

Updating the game state – algorithm

1. *compute the new grid state by applying the neighbourhood rules to the old state*
2. *copy the new grid state into the old grid state*

Should we have given Grid a copy method?

Step 1 refinement

1.1 *for each row i*
1.2 *for each column j*
1.3 *find the number of alive neighbours around (i,j) in the old grid*
1.4 *if number of alive neighbours is 3*
1.5 *set (i,j) in the new grid state to alive*
1.6 *else if number of alive neighbours is 2 and (i,j) is alive in the old grid*
1.7 *set (i,j) in the new grid state to alive*
1.8 *else*
1.9 *set (i,j) in the new grid state to dead*

The updateState method

```
public void updateState() {
   // compute the new grid state by applying the
   // neighbourhood rules to the old state

   Grid newState = new Grid(GRID_SIZE);
   for (int i=1; i<GRID_SIZE-1; i++)
      for (int j=1; j<GRID_SIZE-1; j++) {

         int num = state.aliveNeighbours(i,j);
         newState.setCell(i,j,
             num==3 ||
                (state.getCell(i,j) && num==2) );
      }

   // copy the new grid state into the old grid state
   for (int i=1; i<GRID_SIZE-1; i++)
      for (int j=1; j<GRID_SIZE-1; j++) {
         state.setCell(i,j,newState.getCell(i,j));
}
```

Completed GameOfLife class

```
public class GameOfLife {
   private static final int MAX_GENERATIONS = 200;
   private static final int GRID_SIZE = 500;
   private Grid state;
   private EasyGraphics graphics;
   public GameOfLife(int size) {…}
   public void updateState() {…}
   public void initalise() {…}
   public void display(EasyGraphics g) {…}

   public static void main(String[] args) {
      GameOfLife g = new GameOfLife(300);
      g.initialise();
      for (int i=0; i<MAX_GENERATIONS; i++) {
           g.display();
           g.updateState();
      }
   }
}
```

Looking back

- Could we have done anything better or differently?
- Does the **`updateState`** method belong in **`GameOfLife`**? Should it be in **`Grid`**?
- Or in the potential class we didn't use, **`Cell`**?
- What about a **`copy()`** method for **`Grid`**?

Summary of key points

- We should write classes with **reuse** in mind
- We should provide a **test harness**
- When writing a new class, think first about the **specification** for the class (the data it holds and the operations it provides)
- Only think about the **implementation** issues after that
- Using constants and the **`Object`** class rather than something specific can make classes more general