

SWEN20003
Object Oriented Software Development

Arrays and Strings

Semester 1, 2020

The Road So Far

- OOP Foundations
 - ▶ A Quick Tour of Java
 - ▶ Classes and Objects

Lecture Objectives

After this lecture you will be able to:

- Understand how to use Arrays
- Understand how to use Strings

Motivation

- Store a single integer value

```
int x;
```

- Store two integer values

```
int x1, x2;
```

- Store n integer values

```
int[] intArray;
```

Keyword

Array: A sequence of elements *of the same type* arranged in order in memory

Array Declaration

```
basetype[] varName; OR  
basetype varName[];
```

- **Declares** an array (`[]`)
- Each *element* is of type `basetype`

```
int[] intArray;
```

How many elements does this array have?

Pitfall: Array Declaration

```
int[] intArray;  
int x = intArray[0];
```

Program.java:13: error: variable intArray might not have been initialized

- Arrays must be initialised, just like any other variable
- Let's look at how

Array Initialization and Assignment

```
int[] intArray_1 = {0, 1, 2, 3, 4};
```

- How many elements?
- What are their values?

```
int[] intArray_2 = new int[100];
```

- How many elements?
- What are their values?

```
int[] intArray_1 = new int[n];  
int[] intArray_2 = intArray_1;
```

- How many elements?
- What are their values?

Assess Yourself

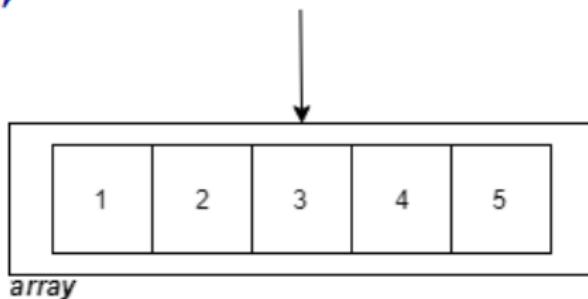
```
int[] intArray_1 = {10, 20, 30, 40};  
int[] intArray_2 = intArray_1;  
  
System.out.println(intArray_2[0]);  
  
intArray_1[0] = 15;  
  
System.out.println(intArray_2[0]);
```

Program Output:

10
15

Pitfall: Array Assignment

```
int [] array = {  
    1, 2, 3, 4, 5  
};
```



- Array is a data type, similar to data types you create by defining
- Arrays are *references*!
- Manipulating one reference affects all references

Assess Yourself

Write a Java static method, `computeDoublePowers`, that accepts an integer `n`, and returns an array of `doubles` of that size. Your method should then fill that array with increasing powers of two (starting from 1.0).

Assess Yourself

```
public static double[] computeDoublePowers(int n) {  
    double[] nums = new double[n];  
  
    for (int i = 0; i < n; i++) {  
        nums[i] = Math.pow(2, i);  
    }  
  
    // For sanity checking  
    for (int i = 0; i < n; i++) {  
        System.out.println(nums[i]);  
    }  
  
    return nums;  
}
```

Multi-Dimensional Arrays

- Java permits “multi-dimensional” arrays
- Technically exist as “array of arrays”
- Declared just like 1D arrays

```
int[][] nums = new int[10][10]; // Square array  
int[][] nums = new int[10][]; // Irregular array
```

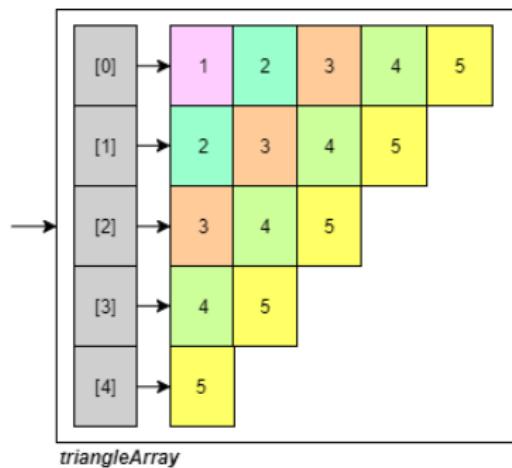
- Initialising 2D arrays slightly more complicated

```
for (int i = 0; i < nums.length; i++) {  
    nums[i] = new int[<length_of_subarray>];  
}
```

Assess Yourself

Write a program that can generate the following 2D array:

```
int[][] triangleArray = {  
    {1, 2, 3, 4, 5},  
    {2, 3, 4, 5},  
    {3, 4, 5},  
    {4, 5},  
    {5},  
};
```



Can you write your program with as **few assumptions as possible?**

Assess Yourself

```
public class IrregularArray {  
    public static void main(String[] args) {  
        final int HEIGHT = 5;  
        final int MAX_WIDTH = HEIGHT;  
  
        int[][] triangleArray= new int[HEIGHT] [] ;  
  
        for (int i = 0; i < HEIGHT; i++) {  
            triangleArray[i] = new int[HEIGHT - i];  
  
            for (int j = 0; j < HEIGHT - i; j++) {  
                triangleArray[i][j] = i + j + 1;  
            }  
        }  
    }  
}
```

Arrays of Objects

Arrays can be used to store objects.

Follow the steps below to create and store objects.

- Declaration of the array:

```
Circle[] circleArray;
```

- Allocation of Storage:

```
circleArray = new Circle[25];
```

- ▶ The above statement created an array that can store references to 25 Circle objects.
- ▶ Circle objects are not created, you have to create them and store them in the array - see next example.

Arrays of Objects - Example

```
// CircleArray.java
class CircleArray{
    public static void main (String[] args){
        //declare an array for Circles
        Circle[] circleArray = new Circle[3];
        // create circle objects and store in array
        for ( int i = 0;  i < circleArray.length; i++) {
            circleArray[i] = new Circle(i,i, i + 2);
        }
        for ( int i = 0;  i < circleArray.length; i++) {
            System.out.println("Circle " + i + " Radius = " +
                circleArray[i].getR());
        }
    }
}
```

Program Output:

```
Circle 0 Radius = 2.0
Circle 1 Radius = 3.0
Circle 2 Radius = 4.0
```

Array Methods

- Indexing

```
int[] intArray = new int[10];
int x = intArray[0]; // The value of x will be 0
int x = intArray[10];// Gives out of bounds error
int x = intArray[-1];// Gives out of bounds error
```

- Length

```
int len = intArray.length
```

- Equality

```
import java.util.Arrays;

int[] n1 = {1, 2, 3};
int[] n2 = {1, 2, 3};

Arrays.equals(n1, n2);
//true if the element values are the same, false otherwise
```

Array Methods

- Resizing - arrays are fixed length; resizing requires creating a new array.

```
int[] intArray = new int[5];  
intArray = new int[intArray.length + 3];
```

- Sorting (“ascending”)

```
Arrays.sort(n1);
```

- Printing

```
System.out.println(Arrays.toString(n1));
```

Output:

```
[1, 2, 3]
```

- Full Array documentation [here](#)

For Each Loop

- More convenient method of iteration
- No indexing required
- Useful when operating with/on the *data*, and not the array

Syntax:

```
for (<type> varName : <iterable object>) {  
    <block of code to execute>  
}
```

Example:

```
for (Circle c : circleArray) {  
    System.out.println(c.getRadius());  
}
```

Strings

Strings

You have already seen Strings in use:

```
public static void main(String[] args) {... }
```

```
final String STRING_CONSTANT = "Welcome to Java";
```

```
public String toString() {....}
```

```
System.out.println("arg[" + i + "]: " + args[i]);
```

But what is a String?

Assess Yourself

A “String” is a(n) what?

- ① Object (not technically correct)
- ② Class
- ③ Variable
- ④ Data Type
- ⑤ Method
- ⑥ Privacy Modifier
- ⑦ I have literally no clue

Strings

- Strings store sequences of characters
- String is a Java class
- Used to represent messages, errors, and “character” related attributes like name
- Incredibly powerful for input and output

Keyword

String: A Java class made up of a sequence of characters.

Strings

Some examples of String variables

```
String s1 = "This is a String";
String s2 = "This is " + "also a String";
String s3 = "10";
String s4 = "s3 is still a string, even though it's a number";
```

Java Strings are almost identical to Python, except you **can't use single quotes**.

Assess Yourself

What does this code output?

```
System.out.println("Game of Thrones season 8 was "good".");
```

- ① "Game of Thrones season 8 was "good"."
- ② Game of Thrones season 8 was "good".
- ③ Game of Thrones season 8 was good.
- ④ Error

Special Characters

- Some characters (like ") are “reserved”
- Mean something special to Java
- Need to “escape” them with “\” to use alternate meaning
- Examples “\n” (newline), “\t” (tab) “\"” (quotation)

```
System.out.println("Game of Thrones season 8 was \"good\".");
```

Keyword

Escaping: To include “special” characters in a string, use “\” to *escape* from that character’s normal meaning.

String Operations

- You can use + (and +=) to append(concatenate two strings
 - ▶ `System.out.println("Hello " + "World");`
 - ▶ Prints "Hello World"
- + is clever: if either operand is a string, it will turn the other into a string
 - ▶ `System.out.println("a = " + a + ", b = " + b);`
 - ▶ If $a = 1$ and $b = 2$, this prints: "a = 1, b = 2"
- Why is this useful?

Assess Yourself

- `System.out.println("1 + 1 = " + 1 + 1);`
- Actually prints "1 + 1 = 11"
- `System.out.println("1 + 1 = " + (1 + 1));`
- Prints "1 + 1 = 2"

Assess Yourself

- Name some “logical” things you might do with a String
- Think about how you would do them in C and Python

String Methods

- Length
 - ▶ C: Need a helper/buddy variable
 - ▶ Python: `len("Hello")`
 - ▶ Java: `"Hello".length()`

- Upper/Lower case
 - ▶ C: `toupper(*s)`
 - ▶ Python: `s.upper()`
 - ▶ Java: `s.toUpperCase()`

- Split
 - ▶ C: `Strtok`
 - ▶ Python: `s.split()`
 - ▶ Java: `s.split(" ")`

String Methods

- Check substring presence
 - ▶ C: Why
 - ▶ Python: "Hell" in s
 - ▶ Java: s.contains("Hell")
- Find substring location
 - ▶ C: Never mind
 - ▶ Python: s.find("Hell")
 - ▶ Java: s.indexOf("Hell")
- Substring
 - ▶ C: I'm out
 - ▶ Python: s[2:7]
 - ▶ Java: s.substring(2, 7)

String Methods

- The full String class documentation can be found [here](#).

Assess Yourself

What does this output?

```
String s = "Hello World";
s.toUpperCase();
s.replace("e", "i");
s.substring(0, 2);
s += " FIVE";
System.out.println(s);
```

"Hello World FIVE"

Immutability

- Strings are *immutable*; once created, they can't be modified, only replaced
- This means that every String operation **returns** a new String
- We learnt about immutability in our previous topic: Classes and Objects - String is an example of a immutable class
- Let's fix up that code

Assess Yourself

What does this output?

```
String s = "Hello World";
s = s.toUpperCase();
s = s.replace("e", "i");
s = s.substring(0, 2);
s += " FIVE";
System.out.println(s);
```

"HE FIVE"

Assess Yourself

What does this output?

```
System.out.println("Hello" == "Hello");
```

true

Assess Yourself

What does this output?

```
String s = "Hello";
System.out.println(s == "Hello");
```

true

Assess Yourself

What does this output?

```
String s = "Hello";
String s2 = "Hello";
System.out.println(s == s2);
```

true

Assess Yourself

What does this output?

```
String s = "Hello";
String s2 = new String("Hello");
System.out.println(s == s2);
```

false

Equality

- In the previous example s and s2 are references to *objects*.
- To check equality between two objects we must use the equals method.
 - ▶ Remember the equals method from our previous topic - a standard method every class should have

```
String s = "Hello";
String s2 = new String("Hello");
System.out.println(s.equals(s2));
```

true

Keyword

.equals: A method used to check two *objects* for equality

Lecture Objectives

Upon completion of this topic you will be able to:

- Use Arrays
- Use Strings

SWEN20003
Object Oriented Software Development

Input and Output

Semester 1, 2020

The Road So Far

- Subject Introduction
- A Quick Tour of Java
- Classes and Objects
- Arrays and String
- Software Tools

Lecture Objectives

Upon completion of this topic you will be able to:

- Accept input to your programs through:
 - ▶ Command line arguments
 - ▶ User input
 - ▶ Files
- Write output from your programs through:
 - ▶ Standard output (terminal)
 - ▶ Files
- Use files to store and retrieve data during program execution
- Manipulate data in files (i.e. for computation)

Input:

Command Line Arguments

Command Line Arguments

Let's take a look back at "Hello World"

```
public static void main(String[] args)
```

What exactly is this?

Command Line Arguments

```
void main(String[] args)
```

- `args` is a variable that stores command line arguments
- `String[]` means that `args` is an *array* of Strings

Entering Arguments - Terminal

- If you compile and run Java from the terminal the syntax is very similar to C.

```
java MyProg Hello World 10
```

- This fills the `args` variable with three elements, "Hello", "World" and "10"
- For multiword Strings, remember to use quotes
- Also note that "10" is a String, not an `int`

```
java MyProg "Hello World" 10
```

- This fills the `args` variable with two elements, "Hello World" and "10"

Entering Arguments - IntelliJ

- Because IDEs do a lot of “behind the scenes” magic, command line arguments are a bit different
- In IntelliJ we have to set the “run configuration” to provide command line arguments
- You can find a walkthrough of the process [here](#)

What Next?

- How do you actually use the arguments once they are put into the program?
- Access the elements of the array by *indexing*
- Identical syntax to accessing array elements in C, or list/tuple elements in Python

```
java MyProg "An" "Argument" "This is another argument"
```

```
System.out.println(args[0]);  
System.out.println(args[1]);  
System.out.println(args[2]);
```

```
"An"  
"Argument"  
"This is another argument"
```

Keyword

Command Line Argument: Information or data provided to a program when it is *executed*, accessible through the `args` variable.

Assess Yourself

Write a program that creates a Person object from three **command line arguments**, and then outputs the object as a String.

A Person is created from three arguments:

- **int** age - age, in years
- **double** height - height, in metres
- **String** name - name, as a String

Assess Yourself

```
public class Program {  
    public static void main(String[] args) {  
        int age = Integer.parseInt(args[0]);  
        double height = Double.parseDouble(args[1]);  
        String name = args[2];  
  
        Person person = new Person(age, height, name);  
        System.out.println(person);  
    }  
}
```

Example input:

```
java Program 27 1.68 "Emily Brown"
```

Example output:

```
"Emily Brown - age: 27, height: 168cm"
```

Assess Yourself

- No interactivity
- Usually for program configuration
- Only when the question tells you! Probably never.
- Let's look at the interactive alternative

Input:
Scanner

Scanner

- Java offers a much more powerful approach to input than C and Python called the Scanner
- We'll look at some of the capabilities, but check out the full documentation [here](#)

Scanner

- Need to import the library first

```
import java.util.Scanner;
```

- Then we create the Scanner

```
Scanner scanner = new Scanner(System.in);
```

- Only ever create **one** Scanner for each program, or bad things happen

Creating a Scanner

```
Scanner scanner = new Scanner(System.in);
```

- The stream/pipe to receive data from, in this case *standard input* (the terminal)

Keyword

System.in: An object representing the *standard input* stream, or the command line/terminal.

Using a Scanner

- Once we've created the Scanner, what do we do with it?
- Scanner has a number of methods used to read data
- The obvious first:

```
String s = scanner.nextLine();
```

- Reads a single line of text, up until a “return” or newline character

Using a Scanner

- But there's more:

```
boolean b = scanner.nextBoolean();  
int i = scanner.nextInt();  
double d = scanner.nextDouble();
```

- Reads a single value that matches the method name (**boolean**, **int**, etc...)

Assess Yourself

```
import java.util.Scanner;

public class TestScanner1 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your input: ");
        double d = scanner.nextDouble();
        String s1 = scanner.next();
        String s2 = scanner.nextLine();

        System.out.format("%3.2f,%s,%s", d, s2, s1);
    }
}
```

Input: 5.2 Hello,World Are there any more words?

Output: 5.20, Are there any more words?,Hello,World

Assess Yourself

```
import java.util.Scanner;

public class TestScanner2 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your input: ");
        double d = scanner.nextDouble();
        float f = scanner.nextFloat();
        int i = scanner.nextInt();

        System.out.format("%3.2f , %3.2f , %3d", d, f, i);
    }
}
```

Input: 5 6.7 7.2

Output: Error

Pitfall: nextXXX

- Scanner does not automatically downcast (i.e. float to int)
- When using `nextXXX`, be sure that the input matches what is expected by your code!

Assess Yourself

```
import java.util.Scanner;

public class TestScanner3 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your input: ");
        double d = scanner.nextDouble();
        String s1 = scanner.nextLine();
        String s2 = scanner.nextLine();

        System.out.format("%3.2f , %s , %s", d, s1, s2);
    }
}
```

Input: 5

6.7

7.2

Output: 5.00, ,6.7

Pitfall: Mixing nextXXX with nextLine

- `nextLine` is the **only** method that “eats” newline characters
- In some cases, you may have to follow `nextXXX` with `nextLine`, if your input is on multiple lines

Other Features

```
scanner.hasNext()  
scanner.hasNextXXX()
```

Keyword

.hasNext: Returns **true** if there is *any* input to be read

Keyword

.hasNextXXX: Returns **true** if the next “token” matches *XXX*

Assess Yourself

Write a program that accepts three **user inputs**, creates an IMDB entry for an Actor, and prints the object:

- String name - the name of a character in a movie/TV show
- double rating - a rating for that character
- String review - a review of that character

Here is an example of the output format:

"You gave Tony Stark a rating of 9.20/10"

"Your review: 'I wish I was like Tony Stark...'"

Assess Yourself

```
public class Actor {  
    public static final int MAX_RATING = 10;  
  
    public String name;  
    public double rating;  
    public String review;  
  
    public Actor(String name, double rating, String review) {  
        this.name = name;  
        this.rating = rating;  
        this.review = review;  
    }  
  
    public String toString() {  
        return String.format("You gave %s a rating of %f/%d\n",  
                            name, rating, MAX_RATING)  
            + String.format("Your review: '%s'", review);  
    }  
}
```

Assess Yourself

```
import java.util.Scanner;

public class TestScanner4 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        String name = scanner.nextLine();

        double rating = scanner.nextDouble();
        scanner.nextLine();

        String comment = scanner.nextLine();

        Actor actor = new Actor(name, rating, comment);
        System.out.println(actor);
    }
}
```

Input:

Reading Files

Reading Files

```
1 import java.io.FileReader;
2 import java.io.BufferedReader;
3 import java.io.IOException;
4
5 public class ReadFile1 {
6     public static void main(String[] args) {
7
8         try (BufferedReader br =
9             new BufferedReader(new FileReader("test.txt"))) {
10
11             String text = null;
12
13             while ((text = br.readLine()) != null) {
14                 System.out.println(text);
15             }
16         } catch (Exception e) {
17             e.printStackTrace();
18         }
19     }
20 }
```

Reading Files - Classes

```
try (BufferedReader br = new BufferedReader(new FileReader("test.txt")))
```

- Creates two objects:
 - ▶ FileReader - A low level file ("test.txt") for simple character reading
 - ▶ BufferedReader - A higher level file that permits reading Strings, not just characters
- **try/catch** exception handling statement - will learn properly under exceptions
- **br** is our file variable

Reading Files - Methods

```
while ((text = br.readLine()) != null)
```

- `br.readLine()`: Reads a single line from the file
- `text =:` Assigns that line of text to a variable
- `!= null`: Then check if anything was actually read

Reading Files - Errors

```
catch (IOException e) {  
    e.printStackTrace();  
}
```

- **catch** - Acts as a safeguard to potential errors, prints an error message if anything goes wrong; more on Exceptions later

Reading Files - Libraries

```
import java.io.FileReader;  
import java.io.BufferedReader;  
import java.io.IOException;
```

- All the classes that make the example go; these make file input possible

Reading Files - Scanner

```
import java.io.FileReader;
import java.util.Scanner;
import java.io.IOException;

public class ReadFile2 {
    public static void main(String[] args) {

        try (Scanner file = new Scanner(new FileReader("test.txt"))) {
            while (file.hasNextLine()) {
                System.out.println(file.nextLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- Works the same as BufferedReader, but allows us to *parse* the text, as well as read it
- Smaller buffer size (internal memory), slower, works on smaller files

Assess Yourself

Write a program that reads a file and counts the number of words in the file.

Assess Yourself

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class WordCount {
    public static void main(String[] args) {

        try (BufferedReader br =
            new BufferedReader(new FileReader("test.txt"))) {
            String text;
            int count = 0;

            while ((text = br.readLine()) != null) {
                String words[] = text.split(" ");
                count += words.length;
            }

            System.out.println("# Words = " + count);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Assess Yourself

Write a program that reads a html file and counts the number of lines that `<h1>` in the file.

Assess Yourself

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class CountHeaders {
    public static void main(String[] args) {

        try (BufferedReader br =
            new BufferedReader(new FileReader("test.html"))) {

            String text;

            int count = 0;

            while ((text = br.readLine()) != null) {
                count = text.contains("<h1>") ? count + 1 : count;
            }

            System.out.println("# Headers: " + count);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Assess Yourself

What does the following program do?

Assess Yourself

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class ReadCSV {
    public static void main(String[] args) {

        try (BufferedReader br =
            new BufferedReader(new FileReader("recipe.csv"))) {
            String text;
            int count = 0;

            while ((text = br.readLine()) != null) {
                String cells[] = text.split(",");

                String ingredient = cells[0];
                double cost = Double.parseDouble(cells[1]);
                int quantity = Integer.parseInt(cells[2]);

                System.out.format("%d %s will cost $%.2f\n", quantity,
                    ingredient, cost*quantity);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Reading CSV files

- CSV = *Comma Separated Value*
- Somewhat equivalent to a spreadsheet
- Usually contains a header row to explain columns
- Example:

Ingredient,Cost,Quantity

Bananas,9.2,4

Eggs,1,6

- **Required knowledge for Projects!**

Output:

Writing Files

File Output

```
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

public class FileWrite1 {
    public static void main(String[] args) {
        try (PrintWriter pw =
            new PrintWriter(new FileWriter("testOut.txt"))) {

            pw.println("Hello World");
            pw.format("My least favourite device is %s and its price is $%d",
                    "iPhone", 100000);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

File Output - Classes

```
try (PrintWriter pw = new PrintWriter(new FileWriter("test.txt")))
```

- Creates two objects:
 - ▶ `FileWriter` - A low level file (`"test.txt"`) for simple character output, used to create...
 - ▶ `PrintWriter` - A higher level file that allows more sophisticated formatting (same methods as `System.out`)
- `try` will automatically close the file once we're done
- `pw` is our file variable

File Output - Methods

```
pw.print("Hello ");
pw.println("World");
pw.format("My least favourite device is %s and its price is $%d",
    "iPhone", 100000);
```

- `pw.print` - Outputs a String
- `pw.println` - Outputs a String with a new line
- `pw.format` - Outputs a String, and allows for format specifiers

File Output - Errors

```
catch (IOException e) {  
    e.printStackTrace();  
}
```

- **catch** - Acts as a safeguard to potential errors, prints an error message if anything goes wrong; more on Exceptions later

File Output - Libraries

```
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;
```

- All the classes that make the example go; these make file output possible

Assess Yourself

What does the following program write to the file?

Assess Yourself

```
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Random;

public class FileWrite2 {
    public static void main(String[] args) {
        final int MAX_NUM = 10000;
        final int ITERATIONS = 1000000;

        Random rand = new Random();

        try (PrintWriter pw =
            new PrintWriter(new FileWriter("testOut2.txt"))) {

            int nums[] = new int[MAX_NUM];

            for (int i = 0; i < ITERATIONS; i++) {
                nums[rand.nextInt(MAX_NUM)] += 1;
            }
            for (int i = 0; i < nums.length; i++) {
                pw.format("%4d: %4d\n", i, nums[i]);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Assess Yourself

What does the following program write to the file?

Assess Yourself

```
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Scanner;

public class FileWrite3 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try (PrintWriter pw =
            new PrintWriter(new FileWriter("test.html"))) {
            pw.println("<h1>The Chronicles of SWEN20003</h1>");

            while (scanner.hasNext()) {
                String text = scanner.nextLine();

                pw.println("<p>" + text + "</p>");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

File Input and Output

- You will **not** be expected to write all of this from memory in the test/exam
- If you are asked to manipulate files, you will be given sufficient scaffold/supporting methods
- For now, all you need to do is practice and understand; we'll talk about assessment closer to the test

Assess Yourself

Implement a rudimentary survey/voting system, by writing a program that continuously expects a single input from the user. This input will be one of three options, in response to the question “Which is your favourite Star Wars trilogy?”

The valid responses are 0 (for the “Original” trilogy), 1 (“New”), and 2 (“The other one”).

Once the input has ended, your program should output the results of the survey, one option per line, as below.

Execution:

```
0  
0  
1  
1  
1  
2
```

```
Original Trilogy: 2
```

```
New Trilogy: 3
```

```
Other Trilogy: 1
```

Assess Yourself

```
import java.util.Scanner;

public class Survey1 {
    public static void main(String[] args) {

        final int N_OPTIONS = 3;

        final int ORIGINAL = 0;
        final int NEW = 1;
        final int OTHER = 2;

        int results[] = new int[N_OPTIONS];

        Scanner scanner = new Scanner(System.in);

        while (scanner.hasNextInt()) {
            int vote = scanner.nextInt();
            results[vote] += 1;
        }

        System.out.println("Original Trilogy: " + results[ORIGINAL]);
        System.out.println("New Trilogy: " + results[NEW]);
        System.out.println("Other Trilogy: " + results[OTHER]);

    }
}
```

Assess Yourself

Follow up:

- What would you do if there were five valid inputs?
- What about n inputs?
- What about allowing the user to **tell you** the options, then getting votes?

Combining Reading and Writing

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.PrintWriter;

public class FileReadWrite {
    public static void main(String[] args) {

        try (BufferedReader br = new BufferedReader(new FileReader("input.txt"));
             PrintWriter pw = new PrintWriter(new FileWriter("output.txt"))) {

            String text;

            while ((text = br.readLine()) != null) {
                pw.println(text.toLowerCase());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Application #1: Data Storage/Retrieval

```
/** Using files to store intermediate data during computation */

final int MAX_DATA = 1000;

try (BufferedReader br = new BufferedReader(new FileReader("input.txt"));
    PrintWriter pw = new PrintWriter(new FileWriter("output.txt", true))) {

    // Recover data from previous run
    String oldData[] = loadPreviousData(br);

    String newData[] = new String[MAX_DATA];

    int count = 0;

    while (magicalComputationNeedsDoing()) {
        newData[count] = magicalComputation(oldData);

        count += 1;

        // Once we do enough computation, store the results just in case
        if (count == MAX_DATA) {
            writeData(pr, newData);
            count = 0;
        }
    }
}
```

Application #2: Data Manipulation

```
/** Using Java to parse/manipulate/convert/etc. files */

try (BufferedReader br = new BufferedReader(new FileReader("input.txt"));
     PrintWriter pw = new PrintWriter(new FileWriter("output.txt"))) {
    String text;

    while ((text = br.readLine()) != null) {
        // Manipulate the input file
        String newText = magicalComputation(text);

        // Write to output file
        pw.println(newText);
    }
}
```

Assess Yourself

- ① Write a program that accepts a filename from the user, which holds the marks for students in SWEN20003. Your program must then process this data, and output a histogram of the results
- ② Extend your program so that it accepts two more inputs for the min and max values for the data
- ③ Extend your program so that it accepts one more input for the width of each “bin” in the histogram

Assess Yourself

```
import java.util.Scanner;
import java.io.File;

public class MarkHist {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter filename: ");
        String filename = scanner.nextLine();

        System.out.print("Enter min value: ");
        int min = scanner.nextInt();
        scanner.nextLine();

        System.out.print("Enter max value: ");
        int max = scanner.nextInt();
        scanner.nextLine();

        System.out.print("Enter bin width: ");
        int width = scanner.nextInt();
        scanner.nextLine();

        int data[] = new int[max-min + 1];

        int total = 0;

        ...
    }
}
```

Assess Yourself

```
try (Scanner file = new Scanner(new File(filename))) {
    // Skip the first line
    file.nextLine();

    while (file.hasNext()) {
        String line[] = file.nextLine().split(",", ",");

        int d = Integer.parseInt(line[1]);

        data[d - min] += 1;
        total += 1;
    }

    ...
}

} catch (Exception e) {
    e.printStackTrace();
}
```

Assess Yourself

```
// Print out graph
for (int i = 0; i < data.length; i += width) {
    int sum = 0;

    // Bundle into *width* sized blocks
    for (int j = 0; j < width && i + j < data.length; j++) {
        sum += data[i+j];
    }

    int percentage = (int) (100 * (1.0 * sum)/total);
    String bar = "";

    if (percentage > 0) {
        bar = String.format("%" + percentage + "s", " ")
            .replace(" ", "=");
    }

    int lower = i + min;
    int upper = lower + width - 1;

    // Print the block
    System.out.format("%03d-%03d: %s\n", lower, upper, bar);
}
```

Assess Yourself

Write a program that takes three inputs from the user:

- **String**, a unit of measurement
- **int**, the number of units
- **String**, an ingredient in a recipe

Your code should write in the following format to a file called "**recipe.txt**":

"- Add 300 grams of chicken"

Bonus Task:

Open the file in “append” mode; this means the file will be added to, rather than overwritten, each time you run your code.

Assess Yourself

Write a program that accepts a **filename** from the user, and then processes that file, recording the frequency with which **words** of different lengths appear.

Assess Yourself

Write a program that accepts a HTML filename from the user, and then takes continuous user input and writes it to the file; essentially a Java based HTML writer.

Bonus #1: add validation to detect valid HTML tags (`<p>`, `<h1>`, etc.).

Bonus #2: add “shortcuts”; for example, entering `{text}` might make *text* automatically bold.

SWEN20003
Object Oriented Software Development

Inheritance and Polymorphism

Semester 1, 2020

The Road So Far

- Subject Introduction
- Java Introduction
- Classes and Objects
- Arrays and Strings
- Software Tools
- Input and Output

Learning Outcomes

Upon completion of this topic you will be able to:

- Use **inheritance** to abstract common properties of classes
- Explain the relationship between a **superclass** and a **subclass**
- Make better use of **privacy** and **information hiding**
- Identify errors caused by **shadowing** and **privacy leaks**, and avoid them
- Describe and use method **overriding**
- Describe the **Object** class, and the properties inherited from it
- Describe what **upcasting** and **downcasting** are, and when they would be used
- Explain **polymorphism**, and how it is used in Java
- Describe the purpose and meaning of an **abstract** class

Introduction and Motivation

A Motivating Example

As a rookie game designer, you want to test your skills by implementing a simple, text-based game of [chess](#).

What classes would you use, and what attributes and methods would they have?

Be sure to use **information hiding** and **access control**.



A Motivating Example

Class: Chess (main)

- Attributes
 - ▶ board
 - ▶ players
 - ▶ isWhiteTurn
- Methods
 - ▶ initialiseGame
 - ▶ isGameOver
 - ▶ getNextMove

A Motivating Example

Class: Player

- Attributes
 - ▶ colour
- Methods
 - ▶ makeMove

A Motivating Example

Class: Board

- Attributes

- ▶ Pawn[]
- ▶ Rook[]
- ▶ Knight[]
- ▶ Bishop[]
- ▶ King
- ▶ Queen

- Methods

- ▶ getNextMove
- ▶ isGameOver

A Motivating Example

Class: Pawn

- Attributes

- ▶ isAlive
- ▶ isWhite
- ▶ currentRow
- ▶ currentColumn

- Methods

- ▶ move
- ▶ isValidMove

A Motivating Example

Class: Rook

- Attributes

- ▶ isAlive
- ▶ isWhite
- ▶ currentRow
- ▶ currentColumn

- Methods

- ▶ move
- ▶ isValidMove

A Motivating Example

Class: Bishop

- Attributes

- ▶ isAlive
- ▶ isWhite
- ▶ currentRow
- ▶ currentColumn

- Methods

- ▶ move
- ▶ isValidMove

A Motivating Example

Class: Knight

- Attributes

- ▶ isAlive
- ▶ isWhite
- ▶ currentRow
- ▶ currentColumn

- Methods

- ▶ move
- ▶ isValidMove

A Motivating Example

Class: Queen

- Attributes

- ▶ isAlive
- ▶ isWhite
- ▶ currentRow
- ▶ currentColumn

- Methods

- ▶ move
- ▶ isValidMove

A Motivating Example

Class: King

- Attributes

- ▶ isAlive
- ▶ isWhite
- ▶ currentRow
- ▶ currentColumn

- Methods

- ▶ move
- ▶ isValidMove

What is the problem?

Why is the design for our Chess game poor?

- Repeated code/functionality, hard to debug
- Doesn't represent the "similarity" /relationship between the pieces
- A lot of work required to implement
- Difficult to extend

Pitfall: Poor Design

Think about how you might implement the Board...

```
public class Board {  
  
    private Pawn[] pawns;  
    private Rook[] rooks;  
  
    ...  
  
}
```

Are you a terrible programmer? No, you're just inexperienced, and have not learnt how to use Inheritance.

Pitfall: Poor Design

How might you implement methods for the game?

```
public void Move(Pawn pawn) {  
    ...  
}  
  
public void Move(Rook rook) {  
    ...  
}  
  
public void Move(Knight knight) {  
    ...  
}
```

Most, if not all, of the code in these methods would be the same.

Inheritance

Keyword

Inheritance: A form of abstraction that permits “generalisation” of similar attributes/methods of classes; analogous to passing genetics on to your children.

Inheritance

Keyword

Superclass: The “parent” or “base” class in the inheritance relationship; provides general information to its “child” classes.

Keyword

Subclass: The “child” or “derived” class in the inheritance relationship; inherits common attributes and methods from the “parent” class.

Inheritance

- Subclass automatically contains all (public/protected) instance variables and methods in the base class
- Additional methods and/or instance variables can be defined in the subclass
- Inheritance allows code to be **reused**
- Subclasses should be “more specific” versions of a superclass

Designing Superclasses and Subclasses

How could we use inheritance in the chess game example?

What properties could be “generalised” across multiple classes?

Inheriting Attributes

How do we Inherit Attributes?

You can see that all attributes for the “Pieces”, Pawn, Rook, Knight, Bishop, King, Queen are common or “general”: `isAlive`, `isWhite`, `currentRow`, `currentColumn`.

So we can define them in a **parent class (Superclass)**, named `Piece`, and all make all other pieces **child classes (Subclasses)** of the `Piece` class.

In the next example, I will only choose two attributes (`currentRow`, `currentColumn`) for demonstration purposes but the concepts can be used for any number of attributes.

Implementing Inheritance

Superclass

```
public class Piece {  
    private int currentRow;  
    private int currentColumn;  
  
    public int getCurrentRow() {  
        return currentRow;  
    }  
    public void setCurrentRow(int currentRow) {  
        this.currentRow = currentRow;  
    }  
    public int getCurrentColumn() {  
        return currentColumn;  
    }  
    public void setCurrentColumn(int currentColumn) {  
        this.currentColumn = currentColumn;  
    }  
}
```

Implementing Inheritance

Subclasses

```
public class Rook extends Piece {  
    public void move(int toRow, int toColumn) { .... }  
  
    public boolean isValidMove(int toRow, int toColumn) { .... }  
}
```

```
public class Knight extends Piece {  
    public void move(int toRow, int toColumn) { .... }  
    public boolean isValidMove(int toRow, int toColumn) {.... }  
}
```

Both the Rook class and the Knight class inherit the attributes in the Piece class although they are not defined in the class itself.

But what does this really mean?

Defining Inheritance

Keyword

extends: Indicates one class **inherits** from another

- Inheritance defines an “**Is A**” relationship
 - ▶ All Rook objects are Pieces
 - ▶ All Dog objects are Animals
 - ▶ All Husky objects are Dogs
- Only use inheritance when this relationship **makes sense**
- A subclass can use attributes in the superclass - let us see how we do this

Creating Objects

```
1 public class InheritanceTester {  
2     public static void main(String[] args) {  
3         Rook rook1 = new Rook();  
4         System.out.println("rook1 location: (" + rook1.getCurrentRow()  
5                             ", " + rook1.getCurrentColumn() + ")");  
6  
7         Piece rook2 = new Rook(); // Rook "is a" Piece  
8         System.out.println("rook2 location: (" + rook2.getCurrentRow()  
9                             ", " + rook2.getCurrentColumn() + ")");  
10  
11        Rook rook3 = new Piece(); // Invalid because a Piece "is not a" Rook  
12    }  
13 }
```

Program Output:

```
rook1 location: (0,0)  
rook2 location: (0,0)
```

Although the getters and setters are in the parent class, the child class could use them, because they were *inherited* from the parent.

Initializing with Constructors

What about the Constructors?

Do we copy and paste parent constructors into subclass constructors?

Of course not!

The keyword **super** can be used to invoke (call) the constructor of the super class.

Keyword

super: Invokes a constructor in the **parent** class

Constructors

```
1  public class Piece {  
2      private int currentRow;  
3      private int currentColumn;  
4      public Piece(int currentRow, int currentColumn) {  
5          this.currentRow = currentRow;  
6          this.currentColumn = currentColumn;  
7      }  
8      public int getCurrentRow() {...}  
9      public void setCurrentRow(int currentRow) {...}  
10     public int getCurrentColumn() {...}  
11     public void setCurrentColumn(int currentColumn) {...}  
12 }
```

```
1  public class Rook extends Piece {  
2      public Rook(int currentRow, int currentColumn) {  
3          super(currentRow, currentColumn);  
4          // Any other code  
5      }  
6      public void move(int toRow, int toColumn) {...}  
7      public boolean isValidMove(int toRow, int toColumn) {...}  
8 }
```

Super Constructor

- May only be used within a subclass constructor
- Must be the first statement in the subclass constructor (if used)
- Parameter **types** to super constructor call must match that of the constructor in the base class

Initializing using Constructors

```
1  public class InheritanceTester {  
2      public static void main(String[] args) {  
3          Rook rook1 = new Rook(2, 10);  
4          System.out.println("rook1 location: (" + rook1.getCurrentRow()  
5                  ", " + rook1.getCurrentColumn() + ")");  
6  
7          Piece rook2 = new Rook(3, 5); // Rook "is a" Piece  
8          System.out.println("rook2 location: (" + rook2.getCurrentRow()  
9                  ", " + rook2.getCurrentColumn() + ")");  
10  
11         Rook rook3 = new Piece(); // Invalid: Piece "is not a" Rook  
12     }  
13 }
```

Program Output:

```
rook1 location: (2,10)  
rook2 location: (3,5)
```

Inheriting and Overriding Methods

How do we Inherit Methods?

Consider the two methods in our Piece class: move and isValidMove

If you consider the logic for implementing the move() method for the Pieces, what do you say?

- Regardless of the Piece, the logic is the same *if your code does not have to check if the new location is valid* (for now, let us assume somebody have validated the new location before calling the method)!

How about the isValidMove() method?

- All pieces must have this method, with the same signature.
- Some of the logic is common: e.g. checking if the new location is not outside the board.
- Some of the logic is different: e.g. the way a Rook can move is different to the way a Knight can move.

Implementing Method Inheritance

Now let us look at how we implement inheritance of methods, `move()` and `isValidMove()` methods.

```
1  public class Piece {  
2      private int currentRow;  
3      private int currentColumn;  
4      // Getters and setters as before, not shown here  
5      public void move(int toRow, int toColumn) {  
6          System.out.println("Piece class: move() method");  
7          this.currentRow = toRow;  
8          this.currentColumn = toColumn;  
9      }  
10     public boolean isValidMove(int toRow, int toColumn) {  
11         System.out.println("Piece class: isValidMove() method");  
12         return true;  
13     }  
14     public String toString() {  
15         return "(" + currentRow + "," + currentColumn + ")";  
16     }  
17 }
```

Implementing Method Inheritance

```
1 public class Rook extends Piece {  
2     public boolean isValidMove(int toRow, int toColumn) {  
3         boolean isValid = true;  
4         System.out.println("Rook class: isValidMove() method");  
5         // Logic for checking valid move and set isVaid  
6         return isValid;  
7     }  
8 }
```

```
1 public class Knight extends Piece {  
2     public boolean isValidMove(int toRow, int toColumn) {  
3         boolean isValid = true;  
4         System.out.println("Knight class: isValidMove() method");  
5         // Logic for checking valid move and set isVaid  
6         return isValid;  
7     }  
8 }  
9
```

Testing Method Inheritance

```
1  public class InheritanceTester {  
2      public static void main(String[] args) {  
3          Rook rook1 = new Rook(2, 10);  
4          if (rook1.isValidMove(4, 10))  
5              rook1.move(4,10);  
6          System.out.println("rook1 location: " + rook1);  
7          System.out.println();  
8  
9          Piece rook2 = new Rook(3, 5);  
10         if (rook2.isValidMove(6, 10))  
11             rook2.move(4,10);  
12         System.out.println("rook2 location: " + rook2);  
13         System.out.println();  
14  
15         Piece rook3 = new Piece(4,6);  
16         if (rook3.isValidMove(8, 12))  
17             rook3.move(8,12);  
18         System.out.println("rook3 location: " + rook3);  
19     }  
20 }
```

Testing Method Inheritance

Program Output:

```
1 Rook class: isValidMove() method
2 Piece class: move() method
3 rook1 location: (4,10)
4
5 Rook class: isValidMove() method
6 Piece class: move() method
7 rook2 location: (6,10)
8
9 Piece class: isValidMove() method
10 Piece class: move() method
11 rook3 location: (8,12)
```

Method Overriding

- When a method is defined only in the parent class (and has the correct visibility which we will discuss later), it gets called regardless of the type of object created (e.g. `move{}` method).
- When a method with the **same signature is defined both in the parent class and the child class**, which method executes purely depends on the **type of object** as opposed to the type of reference (e.g. `isValidMove()` method).
- In the latter case (method defined in both classes), the child class method **Overrides** the method in the parent class.
- Annotation `@Override` can be used in code to indicate that the method is overriding a method in the parent class (optional). See next slide for an example.

Implementing Overridden Methods

```
1  public class Rook extends Piece {  
2  
3  
4     @Override  
5     public boolean isValidMove(int toRow, int toColumn) {  
6         boolean isValid = true;  
7         System.out.println("Rook class: isValidMove() method");  
8         // Logic for checking valid move and set isValid  
9         return isValid;  
10    }  
11  
12 }
```

Note: IDEs support generation of code stubs for overridden methods, and they normally include the `@Override` annotation when generating such code stubs.

Method Overriding

Keyword

Overriding: Declaring a method that exists in a superclass **again** in a subclass, with the **same** signature. Methods can **only** be overridden by subclasses.

Keyword

Overloading: Declaring multiple methods with the same name, but **differing method signatures**. Superclass methods **can** be overloaded in subclasses.

Why Override?

- Subclasses can **extend** functionality from a parent
- Subclasses can **override/change** functionality of a parent
- Makes the *subclass* behaviour **available** when using references of the *superclass* type
- Defines a *general* “interface” in a superclass, with *specific* behaviour implemented in the subclass
 - ▶ This allows seamless access to methods in subclasses using a reference to the superclass - we will see examples later

Extension Through Overriding

Can you improve the design of the `isValidMove` method of your child classes (Rook and Knight)?

Remember, the logic had two parts:

- part that was common to all pieces - checking if the move is within the board
- part that is specific to a particular piece - checking if the move is valid for the particular type of piece

Can we move the generic logic to the parent class and re-use?

Extension Through Overriding - A Better Design

```
1 public class Piece {  
2     final static int BOARD_SIZE = 8;  
3     ...  
4     public boolean isValidMove(int toRow, int toColumn) {  
5         System.out.println("Piece class: isValidMove() method");  
6         return toRow >= 0 && toRow < BOARD_SIZE &&  
7             toColumn >= 0 && toColumn < BOARD_SIZE;  
8     }  
9 }
```

```
1 public class Rook extends Piece {  
2     ...  
3     public boolean isValidMove(int toRow, int toColumn) {  
4         boolean isValid = true;  
5         System.out.println("Rook class: isValidMove() method");  
6         if (!super.isValidMove(toRow, toColumn))  
7             return false;  
8         //Logic for checking valid move and set isValid  
9         return isValid;  
10    }  
11 }
```

Testing Method Inheritance

```
1  public class InheritanceTester {
2      public static void main(String[] args) {
3          Rook rook1 = new Rook(2, 4);
4          if (rook1.isValidMove(4, 10))
5              rook1.move(4,10);
6          System.out.println("rook1 location: " + rook1);
7          System.out.println();
8
9          Piece rook2 = new Rook(3, 5);
10         if (rook2.isValidMove(4, 7))
11             rook2.move(4,7);
12         System.out.println("rook2 location: " + rook2);
13         System.out.println();
14
15         Piece rook3 = new Piece(4,6);
16         if (rook3.isValidMove(8, 12))
17             rook3.move(8,12);
18         System.out.println("rook3 location: " + rook3);
19     }
20 }
```

Testing Method Inheritance

Program Output:

```
Rook class: isValidMove() method  
Piece class: isValidMove() method  
rook1 location: (2,4)
```

```
Rook class: isValidMove() method  
Piece class: isValidMove() method  
Piece class: move() method  
rook2 location: (4,7)
```

```
Piece class: isValidMove() method  
rook3 location: (4,6)
```

Extension Through Overriding

Keyword

super: A reference to an object's parent class; just like **this** is a reference to itself, **super** refers to the attributes and methods of the parent.

Extension Through Overriding - A Better Design

Can we further improve our design to have better encapsulation?

Why should you require the person using your class have to explicitly check if the move is valid?

Can you incorporate this logic into your move method?

```
public class Piece {  
    // attributes and other methods  
  
    public boolean move(int toRow, int toColumn) {  
        System.out.println("Piece class: move() method");  
        if (!isValidMove(toRow, toColumn))  
            return false;  
        this.currentRow = toRow;  
        this.currentColumn = toColumn;  
        return true;  
    }  
    public boolean isValidMove((int toRow, int toColumn) {...}  
}
```

Testing Method Inheritance

```
1  public class InheritanceTester {  
2      public static void main(String[] args) {  
3          Rook rook1 = new Rook(2, 4);  
4          rook1.move(4,10);  
5          System.out.println("rook1 location: " + rook1);  
6          System.out.println();  
7  
8          Piece rook2 = new Rook(3,5);  
9          rook2.move (4,4);  
10         System.out.println("rook2 location: " + rook2);  
11         System.out.println();  
12  
13         Piece rook3 = new Piece(4,6);  
14         rook3.move(8,12);  
15         System.out.println("rook3 location: " + rook3);  
16     }  
17 }
```

Testing Method Inheritance

Program Output:

```
Piece class: move() method
Rook class: isValidMove() method
Piece class: isValidMove() method
rook1 location: (2,4)
```

```
Piece class: move() method
Rook class: isValidMove() method
Piece class: isValidMove() method
rook2 location: (4,4)
```

```
Piece class: move() method
Piece class: isValidMove() method
rook3 location: (4,6)
```

Pitfall: Method Overriding

```
1  public class Piece {  
2      public boolean isValidMove(int currentRow, int currentColumn) {  
3          <block of code to execute>  
4      }  
5  }
```

Overriding can't change return type:

```
1  public class Rook extends Piece {  
2      public int isValidMove(int currentRow, int currentColumn) {  
3          <block of code to execute>  
4      }  
5  }
```

Except when changing to a **subclass** of the original

Recap

Previous Lecture:

- Introduction and Motivation
- Inheriting Attributes
- Inheriting and Overriding Methods

This Lecture:

- Inheritance and Information Hiding
- The Object Class
- Abstract Classes

Inheritance and Information Hiding

Pitfall: Method Overriding

private methods cannot be overridden.

```
1  public class Piece {  
2      private boolean isValidMove(int currentRow, int currentColumn {... }  
3  }
```

```
1  public class Rook extends Piece {  
2      @Override  
3      private boolean isValidMove(int currentRow, int currentColumn) { .. }  
4  }
```

The above definition of the Rook is not valid.

```
1  public class Rook extends Piece {  
2      private boolean isValidMove(int currentRow, int currentColumn) { .. }  
3  }
```

The second definition of the Rook is valid, but does not override the `isValidMove()` method in the `Piece` class - will not get called from a parent class reference and cannot call the parent method with keyword `super`

Restricting Inheritance

If you don't want subclasses to override a method, you can use **final**!

Keyword

final: Indicates that an **attribute**, **method**, or **class** can only be assigned, declared or defined once.

Restricting Inheritance

Keyword

final: Final methods may not be overriden by subclasses.

```
1  public class Piece {  
2      public final boolean move(int toRow, int toColumn) {  
3          System.out.println("Piece class: move() method");  
4          if (!isValidMove(toRow, toColumn))  
5              return false;  
6          this.currentRow = toRow;  
7          this.currentColumn = toColumn;  
8          return true;  
9      }  
10 }
```

This will restrict the `move()` method being overridden.

Access Control

Child classes **cannot** call **private** methods, and **cannot** access **private** attributes of parent classes.

```
1 public class Piece {  
2     private int currentRow;  
3     private int currentColumn;  
4     // Other methods go here  
5 }
```

```
1 public class Rook extends Piece {  
2     public int getCurrentRow() {  
3         return this.currentRow;  
4     }  
5     public void setCurrentRow(int currentRow) {  
6         this.currentRow = currentRow;  
7     }  
8 }
```

The above code for the Rook class is not valid.

Access Control

Child classes **can** call **protected** methods, and **can** access **protected** attributes of parent classes.

```
1 public class Piece {  
2     protected int currentRow;  
3     protected int currentColumn;  
4     // Other methods go here  
5 }
```

```
1 public class Rook extends Piece {  
2     public int getCurrentRow() {  
3         return this.currentRow;  
4     }  
5     public void setCurrentRow(int currentRow) {  
6         this.currentRow = currentRow;  
7     }  
8 }
```

The above code for the Rook class is valid, but see next slide!

Privacy Leaks

Defining attributes as **protected** allows updating them directly from child classes.

However, this should be avoided because it results in **privacy leaks**. The attributes of the parent class should be accessed via **public** or **protected** methods in the parent class.

Example:

- A good design of the Piece class should ensure that any method that updates the attributes, `currentRow`, `currentColumn`, checks if the new position is valid.
- If the attributes are defined as **protected**, the child classes will be able to update the attributes, without doing such checks (checks cannot be enforced by the parent class), resulting in invalid states for the object.

Access Control

Methods in the parent class that are only used by subclasses should be defined as **protected**.

Let us revisit our design for the Piece class.

```
1  public class Piece {  
2      private int currentRow;  
3      private int currentColumn;  
4      final static int BOARD_SIZE = 8;  
5      public Piece(int currentRow, int currentColumn) {...}  
6  
7      public int getCurrentRow() {...}  
8      public void setCurrentRow(int currentRow) {...}  
9      public int getCurrentColumn() {...}  
10     public void setCurrentColumn(int currentColumn) {...}  
11  
12     public final boolean move(int toRow, int toColumn) {...}  
13     public boolean isValidMove(int toRow, int toColumn) {...}  
14     public String toString() {...}  
15 }
```

Is there any method that should be defined as **protected**?

Access Control

The `isValidMove()` method in the `Piece` class is better defined as **protected** because:

- it should be accessed by the child class; and
- should not be accessed directly because the logic is not complete.

```
1  public class Piece {  
2      private int currentRow;  
3      private int currentColumn;  
4      final static int BOARD_SIZE = 8;  
5      public Piece(int currentRow, int currentColumn) {...}  
6  
7      public int getCurrentRow() {...}  
8      public void setCurrentRow(int currentRow) {...}  
9      public int getCurrentColumn() {...}  
10     public void setCurrentColumn(int currentColumn) {...}  
11  
12     public final boolean move(int toRow, int toColumn) {...}  
13     protected boolean isValidMove(int toRow, int toColumn) {...}  
14     public String toString() {...}  
15 }  
16 }
```

Access Control

When overriding a method, a child class cannot further restrict the visibility of an overridden method. When overriding:

- a **public** method in the parent class must remain **public** in the child class
- a **protected** method in the parent class can remain **protected** in the derived or can be made **public**
- a **private** method in the parent class cannot be overridden - as discussed before

Inheritance and Shadowing -Example

```
1 public class PieceS {  
2     public int currentRow;  
3     public int currentColumn;  
4     final static int BOARD_SIZE = 8;  
5     public PieceS(int currentRow, int currentColumn) {  
6         this.currentRow = currentRow;  
7         this.currentColumn = currentColumn;  
8     }  
9     public int getCurrentRow() { return this.currentRow; }  
10    ...  
11 }
```

```
1 public class RookS extends PieceS {  
2     public int currentRow;  
3     public int currentColumn;  
4     public RookS(int currentRow, int currentColumn) {  
5         super(currentRow, currentColumn);  
6     }  
7     public int getCurrentRow() { return this.currentRow; }  
8     ...  
9 }
```

Inheritance and Shadowing -Example

```
1  public class DemoShadowing {
2      public static void main(String[] args) {
3          RookS r1 = new RookS(4,3);
4          System.out.println("r1: row print 1: " + r1.getCurrentRow());
5          System.out.println("r1: row print 2: "+ r1.currentRow);
6
7          PieceS r2 = new RookS(4,3);
8          System.out.println("r2: row print 1: " + r2.getCurrentRow());
9          System.out.println("r2: row print 2: " + r2.currentRow);
10     }
11 }
```

Program Output:

```
r1: row print 1: 0
r1: row print 2: 0
r2: row print 1: 0
r2: row print 2: 4
```

Inheritance and Shadowing

Keyword

Shadowing: When two or more variables are declared with the same name in **overlapping scopes**; for example, in both a subclass and superclass. The variable accessed will depend on the reference type rather than the object.

Don't. Do. It.

You only need to define (common) variables in the superclass.

Privacy Revisited

Keyword

public: Keyword when applied to a class, method or attribute makes it available/visible everywhere (within the class and outside the class).

Keyword

private: Keyword when applied to a method or attribute of a class, makes them only visible within that class. Private methods and attributes are not visible within *subclasses*, and are not inherited.

Keyword

protected: Keyword when applied to a method or attribute of a class, makes them only visible within that class, *subclasses* and also within all classes that are in the same package as that class. They are also visible to *subclasses* in other packages.

Visibility Modifiers

| Modifier | Class | Package | Subclass | Outside |
|------------------------|--------------|----------------|-----------------|----------------|
| <code>public</code> | Y | Y | Y | Y |
| <code>protected</code> | Y | Y | Y | N |
| <code>default</code> | Y | Y | N | N |
| <code>private</code> | Y | N | N | N |

Assess Yourself

Armed with these tools, how would you implement the Board class?

Assess Yourself

```
1  public class Board {  
2      private Piece[][] board;  
3  
4      public boolean makeMove(int fromcurrentRow, int fromcurrentColumn,  
5                               int tocurrentRow, int tocurrentColumn) {  
6          if (board[fromcurrentRow][fromcurrentColumn] == null) {  
7              return false;  
8          }  
9  
10         Piece movingPiece = board[fromcurrentRow][fromcurrentColumn];  
11  
12         if (movingPiece.move(tocurrentRow, tocurrentColumn)) {  
13             board[fromcurrentRow][fromcurrentColumn] = null;  
14             board[tocurrentRow][tocurrentColumn] = movingPiece;  
15             return true;  
16         } else {  
17             return false;  
18         }  
19     }  
20  
21 }
```

Assess Yourself

Implement the text-based chess game! The initial state of the board should be:

```
1      |a|b|c|d|e|f|g|h|
2      -----
3      8|R|N|B|Q|K|B|N|R|
4      -----
5      7|P|P|P|P|P|P|P|P|
6      -----
7      6| | | | | | | |
8      -----
9      5| | | | | | | |
10     -----
11     4| | | | | | | |
12     -----
13     3| | | | | | | |
14     -----
15     2|P|P|P|P|P|P|P|P|
16     -----
17     1|R|N|B|Q|K|B|N|R|
```

Assess Yourself

What classes do you need to write, and where would you write code for the “visualisation” part of the game?

How can you write your solution so that it doesn't matter whether it is a *text-based* or *3D* game?

Assess Yourself

You are a software developer in a swarm robotics laboratory, developing software for *heterogenous* swarms, or swarms with *multiple types* of robots.

A swarm can be an arbitrary combination of ground and aerial robots. Ground robots can be wheeled, bipedal (two legs), or spider-like (many legs). Aerial vehicles can be rotary, or winged.

Create a class design for this scenario, including appropriate use of *inheritance*, with *shared* or *common* attributes/behaviour defined in a superclass, and *specific* behaviour defined in subclasses.

Assess Yourself

Class: Robot

- Attributes
 - ▶ position
 - ▶ orientation
 - ▶ batteryLevel
- Methods
 - ▶ move

Assess Yourself

Class: AerialRobot `extends Robot`

- Attributes
 - ▶ altitude

Assess Yourself

Class: `RotaryRobot` `extends` `AerialRobot`

- Attributes
 - ▶ `numRotors`
- Methods
 - ▶ `move`

Assess Yourself

Class: WingedRobot `extends` AerialRobot

- Attributes
 - ▶ isPushPlane
- Methods
 - ▶ move

Assess Yourself

And so on...

The Object Class

Object

Every class in Java implicitly inherits from the `Object` class

- All classes are of type `Object`
- All classes have a `toString` method
- All classes have an `equals` method
- ... among other (less important) things

The `toString` Method

Consider the `Piece` class without a `toString()` method.

```
1 public class TestInheritace {  
2  
3     public static void main(String[] args) {  
4         Piece rook1 = new Rook(3, 5);  
5         System.out.println("rook1 location: " + rook1);  
6     }  
7  
8 }
```

```
rook1 location: Rook@1540e19d
```

The inherited `toString` method is pretty useless, so we **override** it.

The `toString` Method

Adding a `toString` method to the `Piece` class.

```
1 public class Piece {  
2     ...  
3     @Override  
4     public String toString() {  
5         return "(" + currentRow + "," + currentColumn + ")";  
6     }  
7 }
```

```
1 public class TestInheritace {  
2     public static void main(String[] args) {  
3         Piece rook1 = new Rook(2, 4);  
4         System.out.println("rook1 location: " + rook1);  
5     }  
6 }
```

```
1 rook1 location: (2,4)
```

The equals Method

If have not added an equals method to our Piece class or the Rook class so far.

However, we can still call it because it is defined in the Object class.

```
1  public class TestInheritace {  
2      public static void main(String[] args) {  
3          Piece rook1 = new Rook(2, 4);  
4          Piece rook2 = new Rook(2, 4);  
5          System.out.println(rook1.equals(rook2));  
6      }  
7  }
```

false

The inherited equals method is equally useless (returns false anyway), but **overriding** is a bit more work

The equals Method

What do you think the *signature* would be for equals?

```
1  public class Piece {  
2      public boolean equals(Piece otherPiece) {  
3          <block of code to execute>  
4      }  
5  }
```

Although this works, it really did not override the equals method in the Object class.

Remember that equals is inherited from the Object class has the following signature:

```
public boolean equals(Object otherObject)
```

The equals Method

Note: If you auto generated the code for the Piece class, using IntelliJ IDE you will get the following.

```
1  @Override
2  public boolean equals(Object o) {
3      if (this == o) return true;
4      if (o == null  getClass() != o.getClass()) return false;
5      Piece piece = (Piece) o;
6      return (currentRow == piece.currentRow &&
7              currentColumn == piece.currentColumn);
8  }
9  @Override
10 public int hashCode() {
11     return Objects.hash(currentRow, currentColumn);
12 }
```

This method overrides the equals method in the Object class and is the one you should be using - not really the one we introduced in the topic Classes and Objects! The logic can be replaced, depending on how you want to compare the objects.

Useful Terms

Keyword

getClass: Returns an object of type Class that represents the details of the *calling object's class*.

Keyword

instanceof: An operator that gives **true** if an object A is an instance of the same class as object B, or a class that inherits from B.

```
return new Rook() instanceof Piece; // true  
return new Piece() instanceof Rook; // false
```

Useful Terms

Keyword

Upcasting: When an object of a *child* class is assigned to a variable of an *ancestor* class.

```
Piece p = new Rook(2,3);
```

Keyword

Downcasting: When an object of an *ancestor* class is assigned to a variable of a *child* class. Only makes sense if the underlying object is **actually** of that class. Why?

```
Piece robot = new WingedRobot();
WingedRobot plane = (WingedRobot) robot;
```

Polymorphism

Keyword

Polymorphism: The ability to use objects or methods in many different ways; roughly means “multiple forms”.

Overloading same method with various forms depending on **signature**
(Ad Hoc polymorphism)

Overriding same method with various forms depending on **class**
(Subtype polymorphism)

Substitution using subclasses in place of superclasses (Subtype polymorphism)

Generics defining parametrised methods/classes (Parametric polymorphism, *coming soon*)

Abstract Classes

Assess Yourself

Is there anything *strange* about our Piece class we defined?

- What is a Piece?
- If we create a Piece object, what does that mean?
- Does it make sense to have an object of type Piece?

Abstract

How would this code work?

```
Piece p1 = new Piece();  
p1.move(...)
```

It doesn't!

Some classes aren't meant to be instantiated because they aren't **completely defined**.

Although they are nouns they do not correspond to a real-world entity but is only an **abstraction** to define a class of entities (in this example game pieces such as pawns, rooks etc.).

Abstract Classes

Keyword

Abstract Class: A class that represents common attributes and methods of its subclasses, but that is **missing** some information specific to its subclasses. Cannot be instantiated.

Keyword

Concrete Class: Any class that is not abstract, and has well-defined, specific implementations for all actions it can take.

Abstract Classes

Keyword

abstract: Defines a **class** that is **incomplete**. Abstract classes are “general concepts”, rather than being fully realised/detailed.

```
<visibility> abstract class <ClassName> {  
}
```

```
public abstract class Piece {  
    // Attributes and methods go here  
}
```

Abstract Methods

Keyword

abstract: Defines a superclass method that is common to **all** subclasses, but has no implementation. Each subclass then provides its own implementation through **overriding**.

```
<privacy> abstract <returnType> <methodName>(<arguments>);
```

```
public abstract boolean isValidMove(int toRow, int toColumn);
```

Note: If you make the `isValidMove()` method in the `Piece` class **abstract** (like above), it cannot have any implementation like what we did in our previous example. All the logic has to be implemented in the child classes.

Abstract vs. Concrete

Abstract classes are identical, except:

- **May** have abstract methods - abstract classes can have no abstract methods
- Classes with abstract methods **must** be abstract
- **Cannot** be instantiated
- Represent an **incomplete** concept, rather than a **thing** that is part of a problem

If a class is abstract:

```
public abstract class Piece {  
    // attribute and methods  
}
```

The following definition is not valid:

```
Piece p = new Piece(3,2); // Not valid
```

Types of Inheritance

Inheritance can have multiple levels.

Example:

```
public abstract class Shape {  
    // Attributes and methods go here  
}
```

```
public class Circle extends Shape {  
    // Attributes and methods go here  
}
```

```
public class Rectangle extends Shape {  
    // Attributes and methods go here  
}
```

```
public class GraphicCircle extends Circle {  
    // Attributes and methods go here  
}
```

Types of Inheritance

More generally, there are different forms of inheritance.

- Single inheritance (only one super class)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class, many sub classes)
- Multi-Level inheritance (derived from a derived class)
- Hybrid inheritance (more than two types)
- Multi-path inheritance (inheritance of some properties from two sources).

Notes:

- Java does not support Multiple inheritance, and hence some other forms of inheritance that involves Multiple inheritance, such as Multi-path inheritance.
- Java *sort-of* supports multiple inheritance through Interfaces (our next topic), but it is not quite multiple inheritance

Assess Yourself

You are a software developer in a swarm robotics laboratory, developing software for *heterogenous* swarms, or swarms with *multiple types* of robots.

A swarm can be an arbitrary combination of ground and aerial robots. Ground robots can be wheeled, bipedal (two legs), or spider-like (many legs). Aerial vehicles can be rotary, or winged.

Create a class design for this scenario, including appropriate use of *inheritance*, with *shared* or *common* attributes/behaviour defined in a superclass, and *specific* behaviour defined in subclasses.

Assess Yourself

Class: Robot

- Attributes
 - ▶ position
 - ▶ orientation
 - ▶ batteryLevel
- Methods
 - ▶ move

Assess Yourself

Class: AerialRobot `extends Robot`

- Attributes
 - ▶ altitude

Assess Yourself

Class: `RotaryRobot` `extends` `AerialRobot`

- Attributes
 - ▶ `numRotors`
- Methods
 - ▶ `move`

Assess Yourself

Class: WingedRobot `extends` AerialRobot

- Attributes
 - ▶ isPushPlane
- Methods
 - ▶ move

Assess Yourself

```
public abstract class Robot {  
    public abstract void move(...);  
}
```

```
public abstract class AerialRobot extends Robot {  
}
```

```
public class WingedRobot extends AerialRobot {  
    public void move(...) {  
        <block of code to execute>  
    }  
}
```

Our Powers Combine!

Let's make some magic with our new tools...

Write a program that uses the Robot class and its descendants to create a swarm of robots, either using input from the user, or programmatically.

If we wanted to **control** this swarm, **interact** with it, or have it operate autonomously, what abstractions might we use?

Our Powers Combine!

```
import java.util.Random;

public class RobotApp {

    public static void main(String[] args) {
        final int MAX_ROBOTS = 10;
        Robot[] swarm = new Robot[MAX_ROBOTS];

        for (int i = 0; i < MAX_ROBOTS; i++) {
            swarm[i] = createRobot();
        }
        move(swarm);
    }

    // Continued on next slide
```

Our Powers Combine!

```
private static Robot createRobot() {
    final int NUM_ROBOT_TYPES = 1;
    Random rand = new Random();

    switch(rand.nextInt(NUM_ROBOT_TYPES)) {
        case 0:
            return new WingedRobot();
        case 1:
            return new RotaryRobot();
    }

}

private static void move(Robot[] swarm) {
    for (Robot r : swarm) {
        r.move(new Point());
    }
}
```

Assess Yourself

As a developer for the local zoo, your team has been asked to develop an interactive program for the student groups who often come to visit.

The system allows users to search/filter through the zoo's animals to find their favourite, where they can then read details such as the animal's favourite food, where they're found, and other fun facts.

Your teammates are handling the user interface and mechanics; you've been tasked with designing the data representation that underpins the system.

Using what you know of Object Oriented Design, design a data representation for (some of) the animals in the zoo, making sure to use *inheritance* and *abstract classes* appropriately.

Learning Outcomes

Upon completion of this topic you will be able to:

- Use **inheritance** to abstract common properties of classes
- Explain the relationship between a **superclass** and a **subclass**
- Make better use of **privacy** and **information hiding**
- Identify errors caused by **shadowing** and **privacy leaks**, and avoid them
- Describe and use method **overriding**
- Describe the **Object** class, and the properties inherited from it
- Describe what **upcasting** and **downcasting** are, and when they would be used
- Explain **polymorphism**, and how it is used in Java
- Describe the purpose and meaning of an **abstract** class

SWEN20003
Object Oriented Software Development

Interfaces and Polymorphism

Semester 1, 2020

The Road So Far

- Subject Introduction
- Java Introduction
- Classes and Objects
- Arrays and Strings
- Software Tools
- Input and Output
- Inheritance and Polymorphism

Learning Outcomes

Upon completion of this topic you will be able to:

- Describe the purpose and use of an **interface**
- Describe what it means for a class to **use** an interface
- Describe when it is appropriate to use inheritance vs. interfaces
- Use interfaces and inheritance to achieve powerful abstractions
- Make any class “sortable”

Interfaces

Interfaces

In the last lecture you learnt about **abstract classes**.

Keyword

Abstract Class: A class that represents common attributes and methods of its subclasses, but that is **missing** some information specific to its subclasses. Cannot be instantiated.

Interfaces

Interfaces are **vague**, **distant** relatives of abstract classes:

- Defines an “abstract” entity - can’t be instantiated
- Can only contain **constants** and **abstract methods**
- Defines **behaviours/actions** that are common across a number of classes

Keyword

Interface: Declares a set of constants and/or methods that define the **behaviour** of an object.

Can Do

- All interfaces represent a “**Can do**” relationship
- Classes that implement an interface *can do* all the actions defined by the interface
- Interface names are generally called <...>able, and relate to an action
- For example, classes that implement the <Drivable> interface can all be driven, because they implement the `drive()` method

Defining Interfaces

```
public interface Printable {  
  
    int MAXIMUM_PIXEL_DENSITY = 1000;  
  
    void print();  
  
}
```

- Methods **never** have any code
- All methods are *implied* to be **abstract**
- All attributes are *implied* to be **static final**
- All methods and attributes are *implied* to be **public**

Interfaces

Keyword

interface: Defines an *interface*, rather than a class.

Keyword

implements: Declares that a class implements all the functionality expected by an interface.

Implementing Interfaces

```
public class Image implements Printable {  
    public void print() {  
        <block of code to execute>  
    }  
}
```

```
public class Spreadsheet implements Printable {  
    public void print() {  
        <block of code to execute>  
    }  
}
```

- Concrete classes that *implement* an interface **must** implement **all** methods it defines
- Classes that don't implement all methods must be **abstract**

Default Methods

Classes can be “forced” to have an implementation of a method, that can then be overridden.

```
public interface Printable {  
    default void print() {  
        System.out.println(this.toString());  
    }  
}
```

Keyword

default: Indicates a *standard* implementation of a method, that can be overridden if the behaviour doesn't match what is expected of the implementing class.

Assess Yourself

A person can wear many items of clothing and apparel, but each item can go on a different part of a person's body.

Implement a possible interface for this scenario, as well as one or more implementations of the interface's method(s), such that a hypothetical Person object can “wear clothes”.

Bonus: Why are we using an interface for this?

Assess Yourself

```
public interface Wearable {  
    public void wear();  
}
```

```
public class Clothing implements Wearable {  
  
    private String bodyPart;  
  
    public Clothing(String bodyPart) {  
        this.bodyPart = bodyPart;  
    }  
  
    public void wear() {  
        System.out.format("%s is worn on your %s.\n",  
            this.getClass().getName(), this.bodyPart);  
    }  
}
```

Assess Yourself

```
public class Seatbelt implements Wearable {  
  
    private Car car;  
  
    private boolean isWorn = false;  
  
    public Seatbelt(Car car) {  
        this.car = car;  
    }  
  
    public void wear() {  
        this.isWorn = true;  
        this.car.setCanDrive(true);  
    }  
}
```

Even though Clothing and Seatbelt can both be “worn”, there is no logical relationship between them; they should not be represented using inheritance!

Extending Interfaces

```
public interface Digitisable extends Printable {  
    public void digitise();  
}
```

- Interfaces can be extended just like classes
- Forms the same “Is a” relationship
- Used to add additional, specific behaviour

Sorting

What is sorting?

- Arranging things in **a**n order

What can we sort?

- Any piece of data

How do we sort?

```
Arrays.sort(arrayOfThings);
```

But... How? How does Java know how to arrange Robots? Or Dogs?

String

How does Java sort an array of Strings? Why?

```
String[] strings = new String[]{"dragon",
                               "Jon Snow", "Game of Thrones"};
System.out.println(Arrays.toString(strings));
Arrays.sort(strings);
System.out.println(Arrays.toString(strings));
```

[dragon, Jon Snow, Game of Thrones]

[Game of Thrones, Jon Snow, dragon]

String

Class String

java.lang.Object

java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

Comparable Interface

A class that implements Comparable<ClassName>

- Can (unsurprisingly) be compared with objects of the same class
- Must implement `public int compareTo(<ClassName> object)`
- Can therefore be **sorted** automatically

The general use of <ClassName> will be explained in a later lecture, stay tuned

compareTo

How does it work?

- Defines a method allowing us to **order** objects
- Compares **exactly two** objects, A and B
- B can be a *subclass* of A, as long as they are both Comparable
- Returns a negative integer, zero, or a positive integer if object A (**this**) is “less than”, “equal to”, or “greater than” object B (the argument)

```
public int compareTo(String string) {  
    return this.length() - string.length();  
}
```

compareTo

Worked Example

Assess Yourself

Write a class called RandomNumber that implements the Comparable interface.

Each RandomNumber object should have a single instance variable called number, that is given a random integer when the object is instantiated.

When RandomNumbers are sorted, they should appear in **ascending** order, according to the value of number.

Comparable Interface Example

```
import java.util.Random;
import java.util.Arrays;

public class RandomNumber implements Comparable<RandomNumber> {

    private static Random random = new Random();

    public final int number;

    public RandomNumber() {
        this.number = random.nextInt(100);
    }

    public int compareTo(RandomNumber randomNumber) {
        return this.number - randomNumber.number;
    }

    public String toString() {
        return Integer.toString(this.number);
    }
}
```

Comparable Interface Example

```
public static void main(String args[]) {  
    RandomNumber randomNumbers[] = new RandomNumber[10];  
  
    for (int i = 0; i < randomNumbers.length; i++) {  
        randomNumbers[i] = new RandomNumber();  
    }  
  
    System.out.println(Arrays.toString(randomNumbers));  
  
    Arrays.sort(randomNumbers);  
  
    System.out.println(Arrays.toString(randomNumbers));  
}
```

```
[51, 90, 65, 50, 75, 67, 42, 72, 65, 49]  
[42, 49, 50, 51, 65, 65, 67, 72, 75, 90]
```

Next Level Abstraction

```
public class Spreadsheet extends Document implements Printable,  
    Colourable, Filterable, Comparable<Spreadsheet> {  
    public void print() {  
        <block of code to execute>  
    }  
}
```

- Classes can only *inherit* one class, but can *implement* multiple interfaces
- Inheritance and interfaces work together to build very powerful abstractions that make creating solutions much easier

Multiple Inheritance

- You: “Oh, so we *can* do multiple inheritance in Java!”
- Me: “No, you *can’t*.”
- You: “But you just said classes can implement multiple interfaces?”
- Me: “I did. They are not the same thing.”
- You: “But...”
- Me: “Totally. Different. Things.”

Inheritance is for generalising **shared properties** between **similar classes**; “*is a*”.
Interfaces are for generalising **shared behaviour** between (potentially) **dissimilar classes**; “*can do*”.

Polymorphism

Inheritance

```
Robot robot = new WingedRobot(...);
```

Interfaces

```
Comparable<Robot> comparable = new Robot(...);
```

Subtype polymorphism applies to interfaces!

Assess Yourself

Interface or Inheritance?

- All Dogs can bark.

Both?

Needs more context...

Assess Yourself

Interface or Inheritance?

- All Animals, including Dogs and Cats can make noise.

Inheritance

Assess Yourself

Interface or Inheritance?

- All Animals and Vehicles can make noise.

Interface

Assess Yourself

Interface or Inheritance?

- All classes can be compared with themselves.

Interface

Assess Yourself

Interface or Inheritance?

- All Characters in a game can talk to the Player.

Inheritance

Assess Yourself

Interface or Inheritance?

- Some GameObjects can move, some can talk, some can be opened, and some can attack.

Interface

Interface or Inheritance?

Inheritance:

- Represents *passing shared information* from a *parent* to a *child*
- Fundamentally an “Is a” relationship; a *child is a parent*, plus more; hierarchical relationship
- All Dogs are Animals

Interface:

- Represents the ability of a *class* to *perform an action*
- Fundamentally a “Can do” relationship; a *Comparable* object can be *compared* when sorting
- Strings can be compared and sorted

Metrics

A Student is specified by a first and last name, a student ID, and a list of subjects. When Students are sorted, they should appear in increasing student number order.

A Subject is specified by a name, subject code, and a list of students. When Subjects are sorted, they should appear in order of ascending subject code.

A Course is specified by a name, a course code, a list of (possible) subjects, and a list of students. When Courses are sorted, they should appear in order of ascending course code.

Implement appropriate compareTo methods for each class, and implement the Enrollable interface such that a Student can enrol in both a Subject and a Course.

Lecture Objectives

After this lecture you will be able to:

- Describe the purpose and use of an **interface**
- Describe what it means for a class to **use** an interface
- Describe when it is appropriate to use inheritance vs. interfaces
- Use interfaces and inheritance to achieve powerful abstractions
- Make any class “sortable”

Assess Yourself

You had so much fun in SWEN20003 that you decide to use Bagel to build an RPG.

This game involves a number of characters, like the player, their allies, and their enemies. Some of these characters walk around the world, and some allow the player to interact with them.

There are also other objects in the game like keys, blocks, and power-ups, some of which can be interacted with, some that can be collected/picked up, and some that can be “used” after being picked up.

Some objects in the game have health and can “die”, while some are killed or removed instantly. Other objects can attack and defend other characters.

How would you design this system? What is a good way to *represent* it for **other people** to read?

Assess Yourself

We need a new tool!

SWEN20003
Object Oriented Software Development

Modelling Classes and Relationships

Semester 2, 2019

The Road So Far

- OOP and Java Foundations
- Classes and Objects
- Abstraction
 - ▶ Inheritance
 - ▶ Abstract Classes
 - ▶ Polymorphism
 - ▶ Abstract Classes
 - ▶ Interfaces
- Software Tools

Lecture Objectives

After this lecture you will be able to:

- Identify **classes** and **relationships** for a given problem specification
- Represent classes and relationships in **UML**
- Develop simple **Class Diagrams**

In-class code found [here](#)

Hints for Project 2; pay attention!

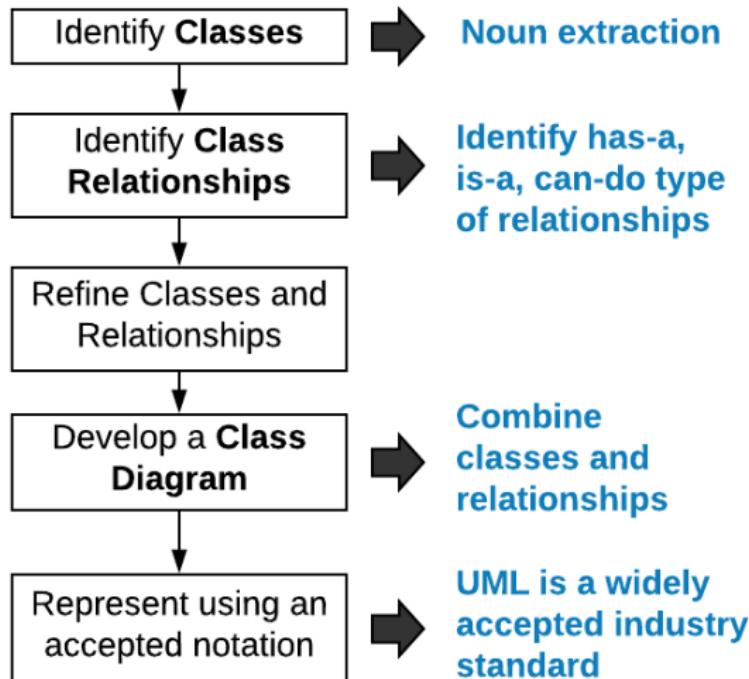
First Steps

Before writing Java code:



Design your system, then **implement** the system (write code) **following** your design!

Design Algorithm



Problem Statement

This game involves a number of characters, like the player, their allies, and their enemies. Some of these characters walk around the world, and some allow the player to interact with them.

There are also other objects in the game like keys, blocks, and power-ups, some of which can be interacted with, some that can be collected/picked up, and some that can be “used” after being picked up.

Some objects in the game have health and can “die”, while some are killed or removed instantly. Other objects can attack and defend other characters.

Identifying Classes - Noun Extraction

This game involves a number of characters, like the player, their allies, and their enemies. Some of these characters walk around the world, and some allow the player to interact with them.

There are also other objects in the game like keys, blocks, and power-ups, some of which can be interacted with, some that can be collected/picked up, and some that can be “used” after being picked up.

Some objects in the game have health and can “die”, while some are killed or removed instantly. Other objects can attack and defend other characters.

Class Design

How would you design this system?

What relationships would the classes have, if any?

Would there be one overarching superclass, or many?

How do we *represent* our design?

Introduction to UML

Keyword

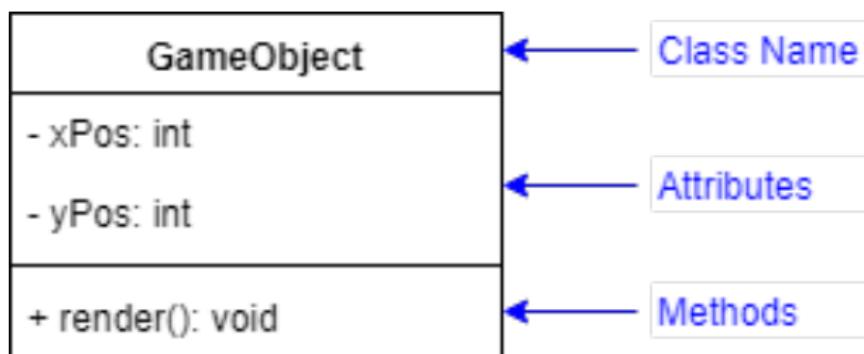
Unified Modeling Language (UML): a graphical modelling language that can be used to represent object oriented analysis, design and implementation.

What you are learning is class modelling; UML is only one notation/tool!

Representing a Class in UML

How would you represent the superclass of the game, GameObject?

- How would you represent the class?
- How would you represent its attributes?
- How would you represent its methods?



Representing A Class

Class

+ attribute1: type = defaultValue

+ attribute2: type

- attribute3: type

+ method1(): return type

- method2(argName: argType, ...): return type

Representing Class Attributes

Components of an attribute:

- Name (e.g. `xPos`)
- Data Type (e.g. : `int`)
- Initial Value (e.g. = `0`)
- Privacy (e.g. `+`)
- Multiplicity

Finite [10] (array)

Range (Known) [1..10] (array or list, etc.)

Range (Unknown) [1..*] (list, etc.)

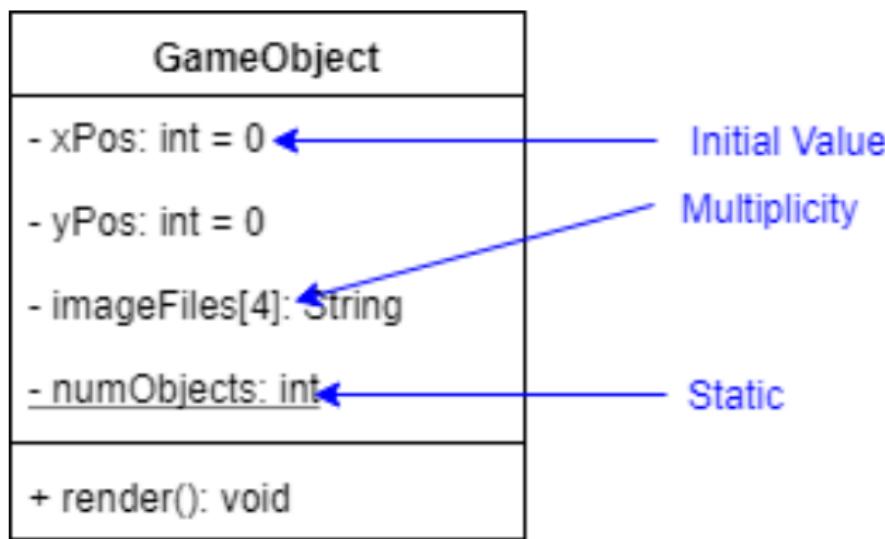
Range (Zero or More) [*] (list, etc.)

- Static (e.g. numMethodCalls : `int`)

Representing Class Attributes

Representing access control:

- + Public
- ~ Package
- # Protected
- Private



Methods

Components of a method:

- Name (e.g. render())
- Privacy (e.g. +)
- Return type (e.g. : **void**)
- Parameters (e.g. name : String)

| GameObject |
|---|
| - xPos: int = 0 - yPos: int = 0 - imageFiles[4]: String <u>- numObjects: int</u> |
| + render(): void + render(imageIndex: int): boolean + getNumObjects(): int |

Parameters

Return type

Static

Assess Yourself

Implement the GameObject class

| GameObject | |
|----------------------------|---------|
| - xPos: int | = 0 |
| - yPos: int | = 0 |
| - imageFiles[4]: | String |
| - numObjects: | int |
| + render(): | void |
| + render(imageIndex: int): | boolean |
| + getNumObjects(): | int |

Assess Yourself

```
public class GameObject {  
  
    private int xPos = 0;  
    private int yPos = 0;  
    private String[] imageFiles;  
    private static int numObjects;  
  
    public void render() {  
        //block of code to execute;  
    }  
  
    public boolean render(int imageIndex) {  
        //block of code to execute  
    }  
  
    public static int getNumObjects() {  
        // block of code to execute;  
    }  
}
```

Representing Class Relationships

Classes may relate to other classes through different types of relationships.

Following are the four common types of relationships between classes:

- Association
- Generalization (Inheritance)
- Realization (Interfaces)
- Dependency

Let's look at how to represent these relationships in UML.

Association

- Represents a **has-a** (containment) relationship between **objects**.
- Allows objects to **delegate** tasks to other objects.
- Shown by a solid line between classes.

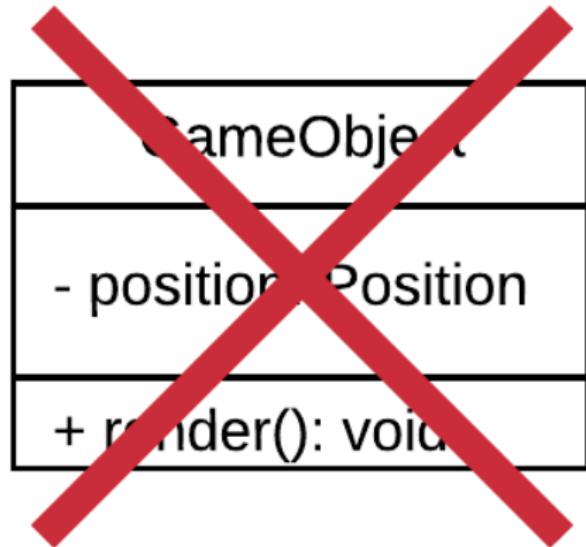
During class refinement the design team decides to represent the position of the `GameObject` with a `Position` class (instead of coordinates).

- Is this a good decision?
- How might the class be represented?
- How does it change our existing design?

Association

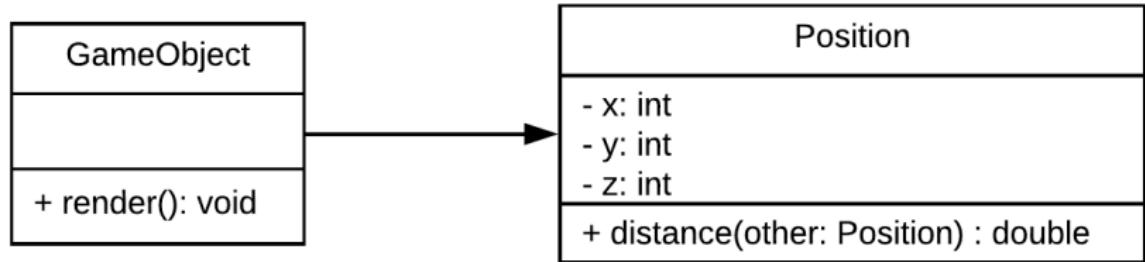
| Position |
|--------------------------------------|
| - x: int - y: int - z: int |
| + distance(other: Position) : double |

Association



Association

When one class is *contained* by another, we always use an **association**.



Keyword

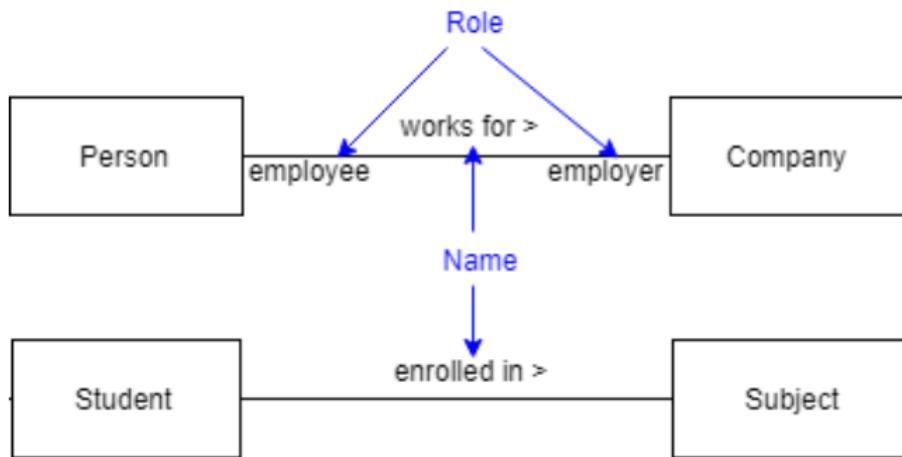
Association: A link indicating one class contains an *attribute* that is itself a class. Does **not** mean one class “uses” another (in a method, or otherwise).

Properties of Association

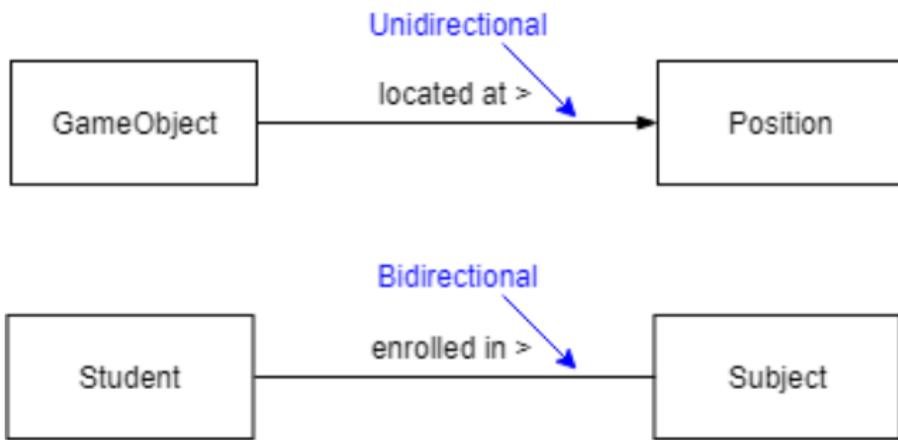
Association links can represent more information:

- Name
- Role labels
- Directionality
 - ▶ Unidirectional (hierarchical, ownership)
 - ▶ Bidirectional (co-operation)
- Multiplicity
- Type
 - ▶ Plain association
 - ▶ Aggregation
 - ▶ Composition

Properties of Association - Name and Role



Properties of Association - Direction



Properties of Association - Multiplicity

Keyword

Multiplicity: specifies the **number of links** that can exist between **instances (objects)** of the associated classes.

- Indicates how many objects of one class relate to one object of another class
- Can be a single number or a range of numbers
- We can add multiplicity on either end of class relationship by simply indicating it next to the class

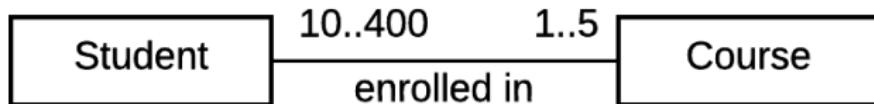
Properties of Association - Multiplicity

- One class can be related to another in the following ways:
 - ▶ One-to-one
 - ▶ One-to-many
 - ▶ One-to-one or more
 - ▶ One-to-zero or one
 - ▶ One-to-a bounded interval (one-to-two through twenty)
 - ▶ One-to-exactly n
 - ▶ One-to-a set of choices (one-to-five or eight)
- Multiplicity can be expressed as:
 - ▶ Exactly one - 1
 - ▶ Zero or one - 0..1
 - ▶ Many - 0..* or *
 - ▶ One or more - 1..*
 - ▶ Exact Number - e.g. 3..4 or 6
 - ▶ Or a complex relationship – e.g. 0..1, 3..4, 6..* would mean any number of objects other than 2 or 5

Assess Yourself

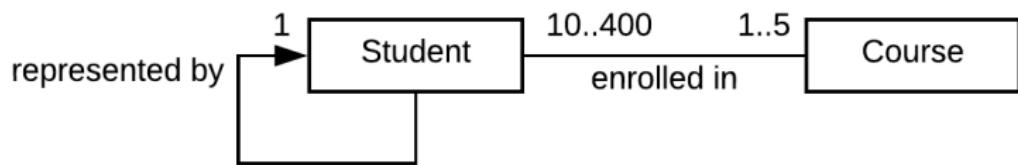
Create a UML representation for the following scenario:

- A Student can take up to five Courses
- A Student has to be enrolled in at least one Course
- A Course can contain up to 400 Students
- A Course should have at least 10 Students



Self-Association

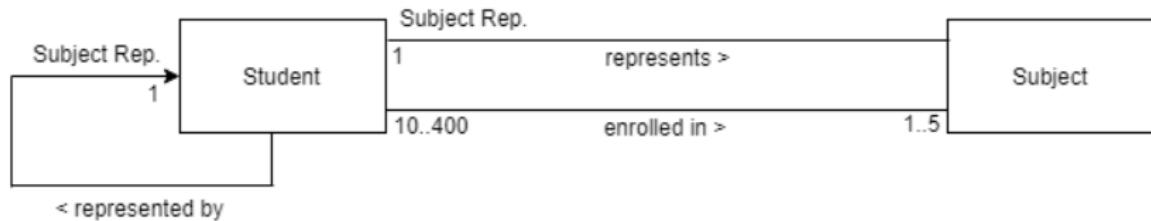
Each Student has a student representative they can contact, who is also a student



Multiple Associations

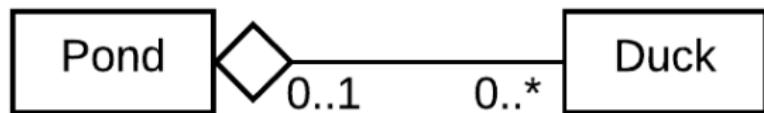
Does the following class relationship enforce a single student rep per-subject?

How about a single student rep per-subject, per-student?



Association - Type (Aggregation)

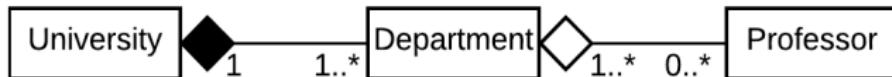
Different form of association, where one class “has” another class, but both exist independently



- If a `GameObject` is destroyed, the `Position` object disappears; dependence
- If the `Pond` object is destroyed, the `Duck` lives on; independence
- Makes sense; a `Duck` can find another `Pond`

Association - Type (Composition)

Different form of association, indicating one class cannot exist without the other; in other words, existing on its own doesn't make sense



- A Department is entirely dependent on a University to exist
- If the University disappears, it makes no sense for a Department to exist
- But a Professor is just a person; they can find another University!

Assess Yourself

What comes next in the design? So far we have:

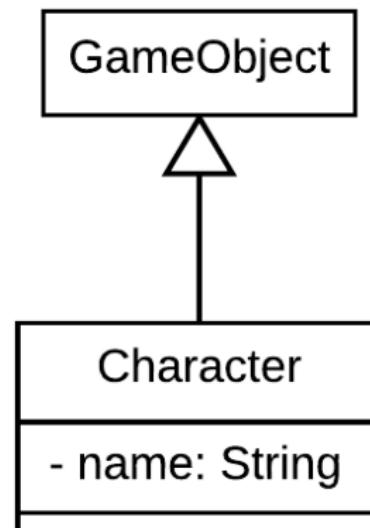
- GameObject
- Position

Assess Yourself

“This game involves a number of characters, like the player, their allies, and their enemies.”

Sounds like **inheritance!**

Generalization - Inheritance

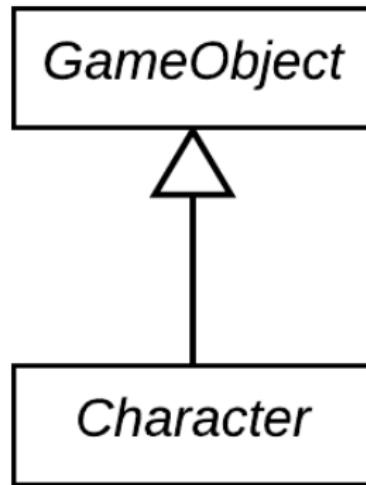


Inheritance

But wait... What happens if I make a `GameObject`? Does that make sense?

Nope!

Abstract Classes



Italicised *methods* or *classes* are **abstract**.

Assess Yourself

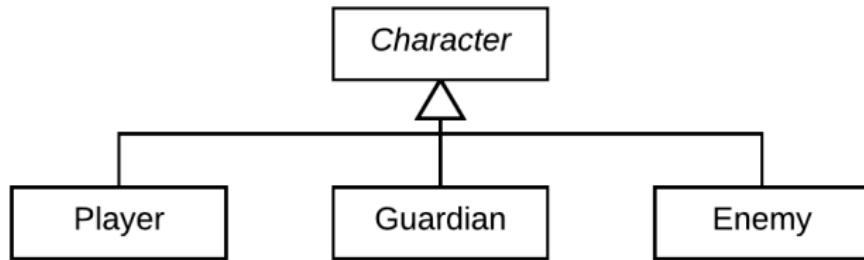
What next? Now we have:

- GameObject
- Position
- Character, which inherits from GameObject

How could we add *behaviour*?

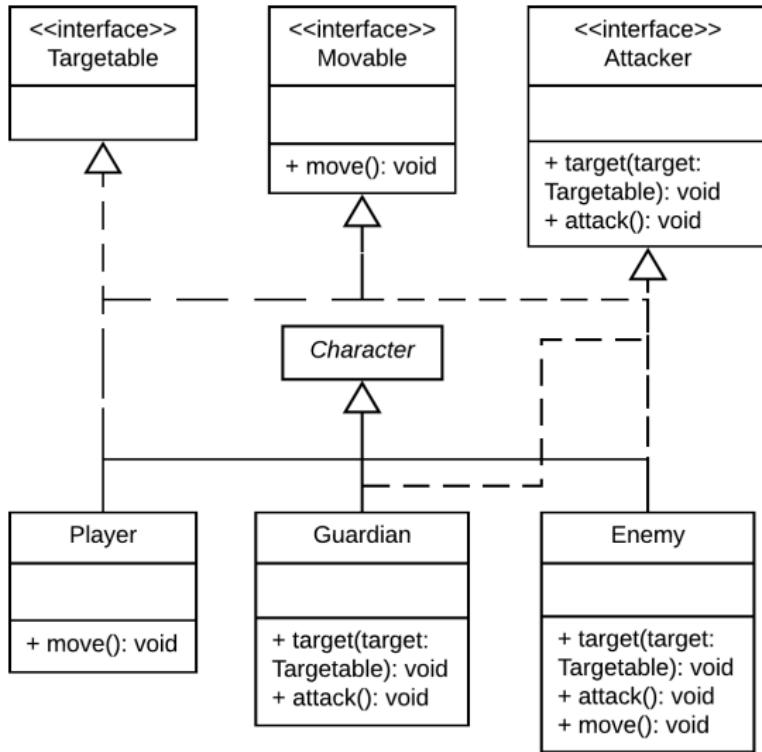
Assess Yourself

Some design help...



Only the Player and Enemy can move. The Guardian and the Enemy can attack each other, and the Enemy can also attack the Player. Both Characters can also target the player.

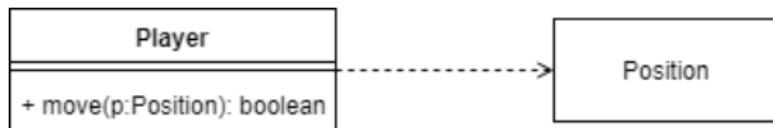
Realization - Interfaces



Dependency

Dependency represents a weak relationship between classes; dependency implies that a change to one class may impact the other class. It is represented by a dotted arrow.

For example, when a method in a class has another class as one of the input parameters, this results in a dependency relationship.

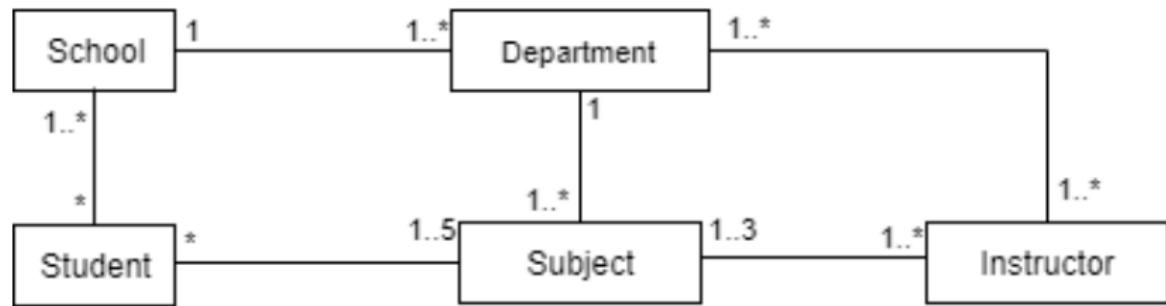


Assess Yourself

Construct a UML diagram for the following description:

- A school has one or more departments.
- A department offers one or more subjects.
- A particular subject will be offered by only one department.
- A department has instructors and instructors can work for one or more departments.
- A student can enrol in up to 5 subjects in a school.
- Instructors can teach up to 3 subjects.
- The same subject can be taught by different instructors.
- Students can be enrolled in more than one school.

Assess Yourself



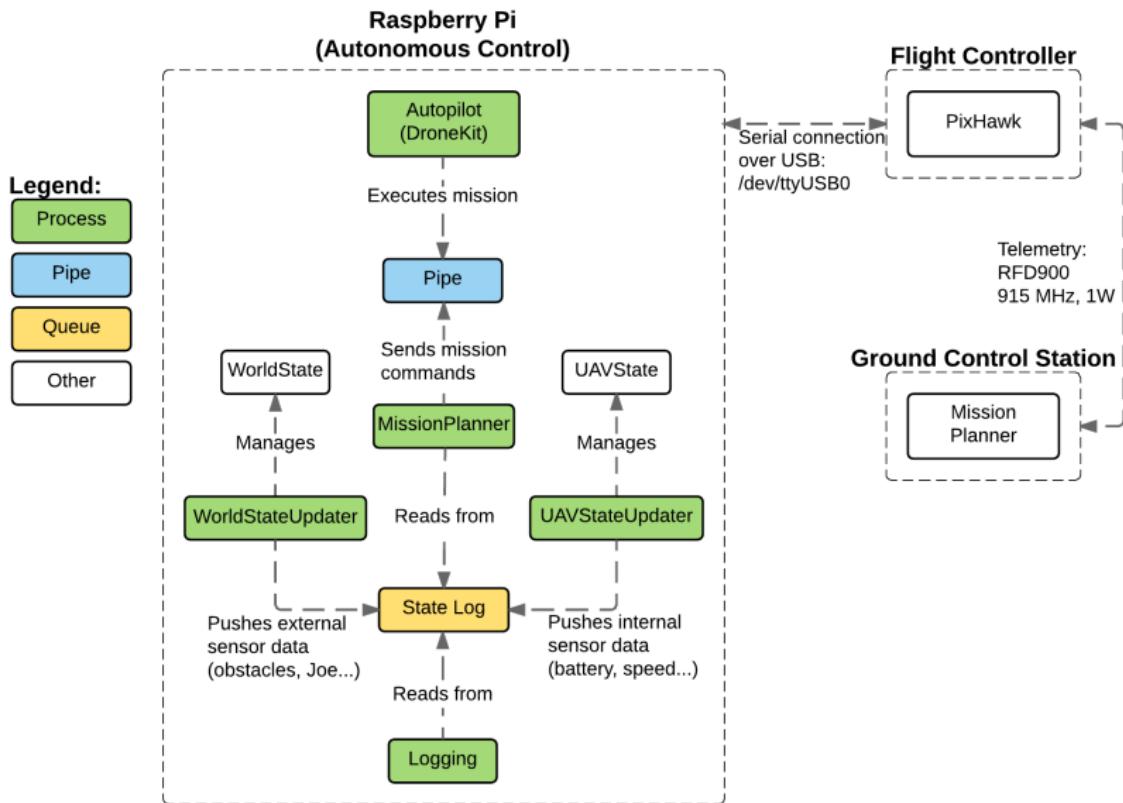
Assess Yourself

How could your Project 1 submission be better designed?

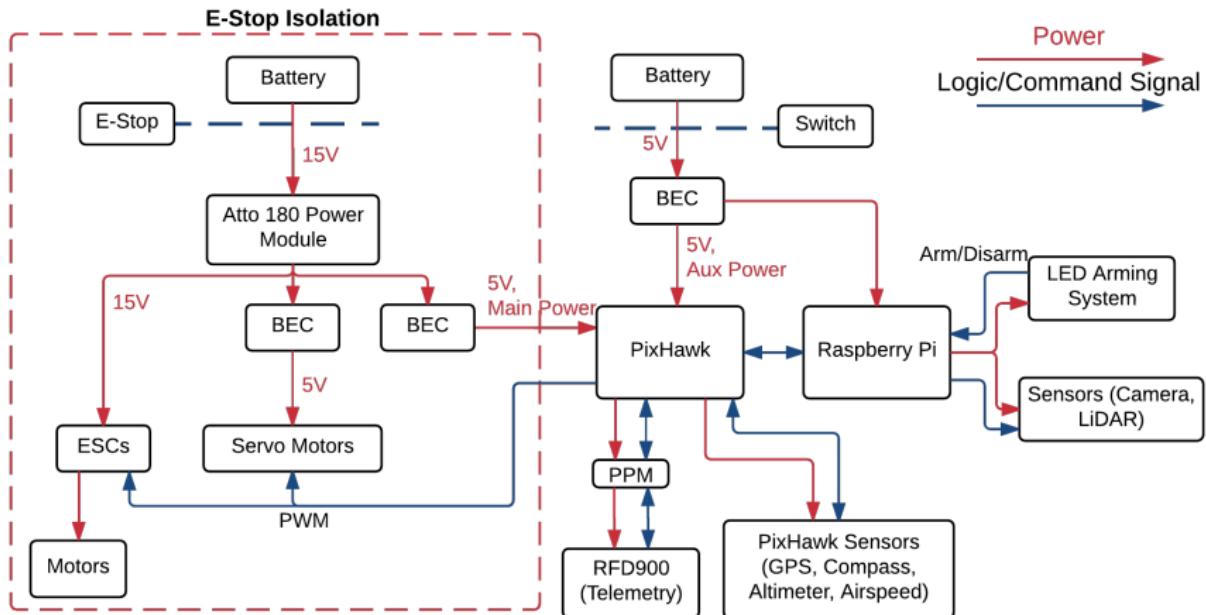
Project answers don't come that easily...

Is modelling useful?

Is Modelling Useful?



Is Modelling Useful?



UML Drawing Tools

- [draw.io](#)
- [LucidChart](#) (Sign up with your [student email](#) for full functionality)
- [StarUML](#)
- IntelliJ with plugins/integrations (useful for converting UML to code instantly)
- And many more...

draw.io is apparently the tool of choice for SWEN30006, but the choice is yours.

Assess Yourself

Finish implementing the class diagram for the game example.

Hint: Good practice for project 2!

Metrics

A world class chef is working to develop a robotic assistant, and they've hired you to build a simulated kitchen to test it out.

The robot needs to be able to:

- Use normal human tools and utensils (oven, fork)
- Open things (cupboards and containers)
- Prepare ingredients (cutting, dicing, boiling)
- Be operated by a human (for safety reasons)

Create a class diagram for this scenario, including any inheritance or realisation (interface) relationships.

SWEN20003

Object Oriented Software Development

Generics I

Semester 2, 2019

The Road So Far

- Java Foundations
- Classes and Objects
 - ▶ Encapsulation
 - ▶ Information Hiding (Privacy)
- Inheritance and Polymorphism
 - ▶ Inheritance
 - ▶ Polymorphism
 - ▶ Abstract Classes
 - ▶ Interfaces
- Modelling classes and relationships

Recap - Modelling classes and relationships

Learning Objectives:

- Identify classes and **relationships**
- Represent classes and relationships in **UML**
- Develop simple **Class Diagrams**

Lecture Objectives

After this lecture you should be able to:

- Understand **generic** classes in Java
- Use **generically typed** classes

Introduction

Java allows class, interface or method definitions to include **parameter types**.

Such definitions are called generics:

- Enables generic logic to be written that applies to any class type
- Allows code re-use

Example: What about a generic Sort class that would allow any type of object to be sorted?

A look back...

You have already seen a generic interface.

A look back...

You have already seen a generic interface.

What is the Comparable interface? How does it work?

A look back...

You have already seen a generic interface.

What is the Comparable interface? How does it work?

```
public interface Comparable<T> {  
    public int compareTo(T other);  
}
```

A look back...

You have already seen a generic interface.

What is the Comparable interface? How does it work?

```
public interface Comparable<T> {  
    public int compareTo(T other);  
}
```

What does T mean?

Type Parameters

- T is a *type parameter*, or type variable

Type Parameters

- T is a *type parameter*, or type variable
- The value of T is literally a type (class/interface); Integer, String, Robot, Book, Driveable

Type Parameters

- T is a *type parameter*, or type variable
- The value of T is literally a type (class/interface); Integer, String, Robot, Book, Driveable
- When T is given a value (type), every instance of the *placeholder* variable is replaced

Type Parameters

- T is a *type parameter*, or type variable
- The value of T is literally a type (class/interface); Integer, String, Robot, Book, Driveable
- When T is given a value (type), every instance of the *placeholder* variable is replaced

```
public class Robot implements Comparable<Robot> {...}  
public class Book implements Comparable<Book> {...}  
public class Dog implements Comparable<Dog> {...}
```

Type Parameters

How do you write a class that can be compared with an object of the same type?

Type Parameters

How do you write a class that can be compared with an object of the same type?

```
public class Dog implements Comparable<Dog> {

    private String name;

    public Dog(String name) {
        this.name = name;
    }

    public int compareTo(Dog dog) {
        return this.name.compareTo(dog.name);
    }
}
```

Type Parameters

Using type parameters allows us to define a class or method that uses arbitrary, **generic** types, that applies to **any** and **all** types.

Type Parameters

Using type parameters allows us to define a class or method that uses arbitrary, **generic** types, that applies to **any** and **all** types.

But why?

Can we compare objects without using the generic Comparable interface?

Using the Non-generic Comparable Intf.

```
public class Circle implements Comparable {
    private double centreX = 0.0, centreY = 0.0;
    private double radius = 0.0;
    @Override
    public int compareTo(Object o) {
        Circle c = null;
        if (o instanceof Circle) {
            c = (Circle)o;
            if (c.radius > this.radius)
                return 1;
            else if (c.radius < this.radius)
                return -1;
            else
                return 0;
        } else {
            return -2;
        }
    }
}
```

Using the Non-generic Comparable Intf.

```
public class Square implements Comparable{
    private double centreX = 0.0, centreY = 0.0;
    private double length = 0.0;
    @Override
    public int compareTo(Object o) {
        Square s = null;
        if (o instanceof Square) {
            s = (Square)o;
            if (s.length > this.length)
                return 1;
            else if (s.length < s.length)
                return -1;
            else
                return 0;
        } else {
            return -2;
        }
    }
}
```

Using the Non-generic Comparable Intf.

```
public class CompareShapes {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(0.0, 0.0, 5);  
        Circle c2 = new Circle(0.0, 0.0, 10);  
        System.out.println("Compare c1 and c2  
                           = " + c1.compareTo(c2));  
        Square s = new Square(0.0, 0.0, 10);  
        System.out.println("Compare c1 and s  
                           = " + c1.compareTo(s));  
    }  
}
```

Using the Non-generic Comparable Intf.

```
public class CompareShapes {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(0.0, 0.0, 5);  
        Circle c2 = new Circle(0.0, 0.0, 10);  
        System.out.println("Compare c1 and c2  
                           = " + c1.compareTo(c2));  
        Square s = new Square(0.0, 0.0, 10);  
        System.out.println("Compare c1 and s  
                           = " + c1.compareTo(s));  
    }  
}
```

Yes it works, but the solution is not elegant!

Using the Non-generic Comparable Intf.

```
public class CompareShapes {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(0.0, 0.0, 5);  
        Circle c2 = new Circle(0.0, 0.0, 10);  
        System.out.println("Compare c1 and c2  
                           = " + c1.compareTo(c2));  
        Square s = new Square(0.0, 0.0, 10);  
        System.out.println("Compare c1 and s  
                           = " + c1.compareTo(s));  
    }  
}
```

Yes it works, but the solution is not elegant!

The programmer has to check for -2 which is not a valid comparison.

Using the Non-generic Comparable Intf.

```
public class CompareShapes {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(0.0, 0.0, 5);  
        Circle c2 = new Circle(0.0, 0.0, 10);  
        System.out.println("Compare c1 and c2  
                           = " + c1.compareTo(c2));  
        Square s = new Square(0.0, 0.0, 10);  
        System.out.println("Compare c1 and s  
                           = " + c1.compareTo(s));  
    }  
}
```

Yes it works, but the solution is not elegant!

The programmer has to check for -2 which is not a valid comparison.

Can we avoid this?

Using the Generic Comparable Intf.

```
public class CircleT implements Comparable<CircleT> {
    private double centreX = 0.0;
    private double centreY = 0.0;
    private double radius = 0.0;

    @Override
    public int compareTo(CircleT c) {
        if (c.radius > this.radius)
            return 1;
        else if (c.radius < this.radius)
            return -1;
        else
            return 0;
    }
}
```

Assume you also have a SquareT class which implements the generic Comparable interface.

Using the Generic Comparable Intf.

```
public class CompareShapesT {  
    public static void main(String[] args) {  
        CircleT c1 = new CircleT(0.0, 0.0, 5);  
        CircleT c2 = new CircleT(0.0, 0.0, 10);  
        System.out.println("Compare c1 and c2 = "  
                           + c1.compareTo(c2));  
        SquareT s = new SquareT(0.0, 0.0, 10);  
        System.out.println("Compare c1 and s = "  
                           + c1.compareTo(s));  
        //The line above will give a compiler error  
    }  
}
```

Using the ArrayList Class

What are the limitations of array?

Using the ArrayList Class

What are the limitations of array?

- Finite length
- Resizing is a manual operation
- Requires effort to “add” or “remove” elements

Using the ArrayList Class

What are the limitations of array?

- Finite length
- Resizing is a manual operation
- Requires effort to “add” or “remove” elements

Is there an alternative?

Using the ArrayList Class

What are the limitations of array?

- Finite length
- Resizing is a manual operation
- Requires effort to “add” or “remove” elements

Is there an alternative?

ArrayList class, which is a generic class, solves the above problems!

Using the ArrayList Class

```
import java.util.ArrayList;
public class PrintCircleRadius {
    public static void main(String[] args) {
        ArrayList<CircleT> circles = new ArrayList<CircleT>();
        circles.add(new CircleT(0.0, 0.0, 5));
        circles.add(new CircleT(0.0, 0.0, 10));
        circles.add(new CircleT(0.0, 0.0, 7));
        printRadius(circles);
    }
    private static void printRadius(ArrayList<CircleT> circles){
        int index = 0;
        for(CircleT c: circles) {
            System.out.println("Radius at index " + index +
                               " = " + c.getRadius());
            index++;
        }
    }
}
```

Using the ArrayList Class

So what does the ArrayList give you?

- Can be iterated like arrays (for-each)
- Automatically handles resizing
- Can *insert*, *remove*, *get*, and *modify* elements at any index (plus many more capabilities)
- Inherently able to `toString()`
- Can't be **indexed** (`[]`)

ArrayList is a class with an *array* as an instance variable.

Using the ArrayList Class

Are there any limitations of the ArrayList class?

Using the ArrayList Class

Are there any limitations of the ArrayList class?

- Although an ArrayList grows automatically when needed, it does not shrink automatically, hence can consume more memory than required - `trimToSize()` method must be invoked to release the excess memory.
- Cannot store primitive data types (int, float, etc.).

Using the ArrayList Class

Elements of an ArrayList can be easily sorted if:

Using the ArrayList Class

Elements of an ArrayList can be easily sorted if:

The stored element class implements the Comparable<T> interface!

The compareTo() method of the class must provide a comparison (returning an integer) which will be used to decide how the elements are sorted.

Using the ArrayList Class

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

public class PrintCircleRadiusSorted {
    public static void main(String[] args) {
        ArrayList<CircleT> circles = new ArrayList<CircleT>();
        circles.add(new CircleT(0.0, 0.0, 5));
        circles.add(new CircleT(0.0, 0.0, 10));
        circles.add(new CircleT(0.0, 0.0, 7));
        Collections.sort(circles);
        printRadius(circles);
    }

    private static void printRadius(ArrayList<CircleT> circles){
        int index = 0;
        for(CircleT c: circles) {
            System.out.println("Radius of circle: at index " +
                               index++ + " = " + c.getRadius());
        }
    }
}
```

Using the ArrayList Class

ArrayList can be used for storing different types of objects, provided they inherit the same base class - therefore not quite different types of objects theoretically.

Why is this useful?

Common behaviour across objects can be executed seamlessly - see next example.

Using the ArrayList Class

```
public abstract class Shape {  
    public abstract double getArea();  
}  
  
public class Circle extends Shape {  
    private double radius = 0.0;  
    // Code for constructors, getter and setter go here  
    @Override  
    public double getArea() {  
        return Math.PI*radius*radius;  
    }  
}  
  
public class Square extends Shape {  
    private double length = 0.0;  
    // Code for constructors, getter and setter go here  
    @Override  
    public double getArea() {  
        return length*length;  
    }  
}
```

Using the ArrayList Class

```
import java.util.ArrayList;

public class ComputeAreaShapes {
    public static void main(String[] args) {
        ArrayList<Shape> shapes = new ArrayList<Shape>();
        shapes.add(new Circle(0.0, 0.0, 5));
        shapes.add(new Circle(0.0, 0.0, 10));
        shapes.add(new Square(0.0, 0.0, 7));
        printArea(shapes);
    }

    private static void printArea(ArrayList<Shape> shapes) {
        int index = 0;
        for(Shape s: shapes) {
            System.out.println("Area of shape: at index " +
                               index++ + " = " + s.getArea());
        }
    }
}
```

Assess Yourself

Implement the method

`ArrayList<Integer> generateList(Scanner scanner)`

which continually accepts integers from a user until they end input, and returns an `ArrayList` of all the integers entered.

Implement a second method

`double average(ArrayList<Integer> numbers)`

that returns the average of all elements in `numbers`.

Assess Yourself

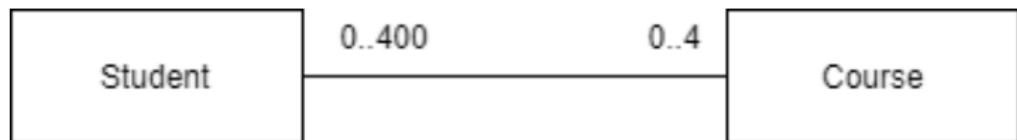
```
public ArrayList<Integer> generateList(Scanner scanner) {  
  
    ArrayList<Integer> numbers = new ArrayList<>();  
  
    while (scanner.hasNextInt()) {  
        numbers.add(scanner.nextInt());  
    }  
  
    return numbers;  
  
}
```

Assess Yourself

```
public double average(ArrayList<Integer> numbers) {  
  
    double sum = 0.0;  
  
    for (Integer number : numbers) {  
        sum += number;  
    }  
  
    return sum / numbers.size();  
}
```

Assess Yourself

Now we can implement this relationship!



Implement the Student and Course classes, ignoring all instance variables but what the diagram shows.

Include methods to enrol a Student in a Course, and vice versa.

Assess Yourself

```
public class Student {  
  
    public static final int MAX_COURSES = 4;  
  
    private ArrayList<Course> courses = new ArrayList<>();  
  
    public boolean addCourse(Course course) {  
        if (courses.size() < MAX_COURSES &&  
            !courses.contains(course)) {  
            courses.add(course);  
            course.addStudent(this);  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

Assess Yourself

```
public class Course {  
  
    public static final int MAX_STUDENTS = 400;  
  
    private ArrayList<Student> students = new ArrayList<>();  
  
    public boolean addStudent(Student student) {  
        if (students.size() < MAX_STUDENTS &&  
            !students.contains(student)) {  
            students.add(student);  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

Assess Yourself

Think carefully... Is there something odd about our design? An abstraction we've missed?

Assess Yourself

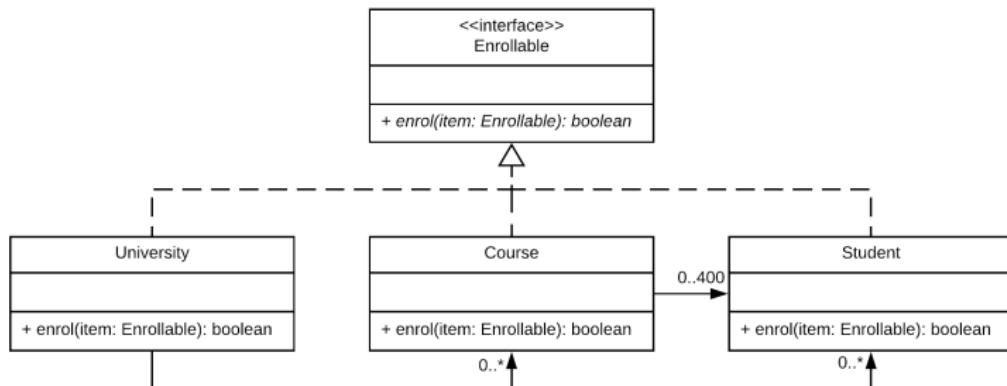
Think carefully... Is there something odd about our design? An abstraction we've missed?

How could we make adding or changing features easier?

Assess Yourself

Think carefully... Is there something odd about our design? An abstraction we've missed?

How could we make adding or changing features easier?



What does this look like?

Assess Yourself

```
import java.util.ArrayList;

public class Student implements Enrollable<Course> {
    public static final int MAX_COURSES = 5;

    ArrayList<Course> courses = new ArrayList<>();

    public boolean enrol(Course course) {
        if (courses.size() < MAX_COURSES &&
            !courses.contains(course)) {
            courses.add((Course)course);
            course.enrol(this);
            System.out.println("Added course successfully");
            return true;
        } else {
            System.out.println("Failed to add course");
            return false;
        }
    }
}
```

Assess Yourself

Write a main method that allows a user to continually enter integers, where each number is added to an ArrayList.

The program should then create a PriorityQueue for arranging the numbers based on the following rules:

- Ascending order
- Descending order
- Shortest (least characters) first
- Ascending order by the last (rightmost) digit

Learning Outcomes

You should be able to:

- Understand **generic** classes in Java
- Use **generically typed** classes

SWEN20003
Object Oriented Software Development

Generics

Semester 1, 2002

Recap & today's lecture - Generics

Recap on last lecture:

- Comparable<T> interface
- How to use ArrayList<T>
- T is a type parameter, can be given to a class. How about generics for a function?.
- Why generics? Well, to allow code-reuse

Today's lecture:

- Defining generics
- Wildcard and subtyping

Defining a Generic Class

Keyword

Generic Class: A class that is defined with an arbitrary type for a field, parameter or return type.

- The type parameter is included in angular brackets after the class name in the class definition heading.
- A type parameter can have any reference type (i.e., any class type) plugged in.
- Traditionally, a single uppercase letter is used for a type parameter, but any non-keyword identifier may be used.
- A class definition with a type parameter is stored in a file and compiled just like any other class.

Defining Generics

```
public class Sample<T> {  
  
    private T data;  
  
    public void setData(T data) {  
        this.data = data;  
    }  
  
    public T getData() {  
        return data;  
    }  
}
```

Defining a Generic Class - Multiple Types

```
public class TwoTypePair<T1, T2> {
    private T1 first;
    private T2 second;

    public TwoTypePair() {
        first = null;
        second = null;
    }

    public TwoTypePair(T1 first, T2 second) {
        this.first = first;
        this.second = second;
    }

    public void setFirst(T1 first){
        this.first = first;
    }

    public void setSecond(T2 second) {
        this.second = second;
    }
    // Additional methods go here
}
```

Using a Generic Class - Multiple Types

```
import java.util.Scanner;
public class TwoTypePairDemo {
    public static void main(String[] args) {

        TwoTypePair<String, Integer> rating =
            new TwoTypePair<String, Integer>("The Car Guys", 8);

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Our current rating for " +
                           rating.getFirst() + " is " + rating.getSecond());
        System.out.println("How would you rate them?");

        int score = keyboard.nextInt();
        rating.setSecond(score);
        System.out.println("Our new rating for "+
                           rating.getFirst() + " is " + rating.getSecond());
    }
}
```

Subtyping

Keyword

Generics' subtyping: Generic classes or interfaces are not related merely because there is a relationship between their type parameters.

```
abstract class Animal {}
class Dog extends Animal {}
class Bear extends Animal {}

Dog dog = new Dog();
Animal a = dog; // OK
List<Dog> dogs = new ArrayList<Dog>();
List<Animal> animals = new ArrayList<Animal>();

dogs = animals; (1) // Compile-time error
animals = dogs; (2) // Compile-time error
```

Subtyping

```
abstract class Animal {}
class Dog extends Animal {}
class Bear extends Animal {}

Dog dog = new Dog();
Animal a = dog; // OK
List<Dog> dogs = new ArrayList<Dog>();
List<Animal> animals = new ArrayList<Animal>();
animals.add(new Bear()); // Add a Bear in to animals

dogs = animals; (1) // Assume this is OK, then ...
Dog dog = dogs.get(0) // Opps, we should get a dog out of dogs,
                     // but what we actually get is a bear.
```

- (1) is not allowed because doing so allows to pull out elements in animals which can be of either type Bear or Dog. But the dogs list is supposed to only contain Dog elements.

Wildcard and subtyping

```
abstract class Animal {}
class Dog extends Animal {}
class Bear extends Animal {}

List<Dog> dogs = new ArrayList<Dog>();
List<Animal> animals = new ArrayList<Animal>();

animals = dogs; (2) // Assume this is OK, then ...
animals.add(new Bear()); // Add a bear to animals. Opps, this in
                        // turn adds a bear to dogs because of
                        // object reference at (2).
```

(2) is not allowed because doing so allows to **add/write** new elements in animals which in turn will add the elements to dogs via the object reference. The new elements added to animals can be of either type Bear or Dog. But the dogs list is supposed to only contain Dog elements.

Wildcard

- Although Dog is a subtype of Animal, the generic collection List<Dog> is not a subtype of List<Animal>. In fact, the common parent of them is List<?>, wherein the ? is a wildcard that stands for **unknown** type.
- Generic wildcards target two primary needs: reading from and inserting to a generic collection.
- Three ways to define a collection using generic wildcards.

```
List<?> listUnknown = new ArrayList<A>();  
List<? extends A> listUnknown = new ArrayList<A>();  
List<? super A> listUnknown = new ArrayList<A>();
```

The Unknown Wildcard

- The `List<?>` means a list is typed to an unknown type. This could be a list of any type.
- Since we do not know the type of the list, we can only read the collection.

```
public void print(List<?> myList){  
    for(Object o: myList){ // o could be of any type  
        System.out.println(o);  
    }  
}  
  
List<Dog> dogs = new ArrayList<Dog>();  
print(dogs);  
List<Bear> bears = new ArrayList<Bear>();  
print(bears);
```

- This can be thought of as read-only collection

The Extends Wildcard

- The `List<? extends A>` means a list of objects that are instances of the class A, or subclasses of A.
- Since we do know the type of elements of the list is A or subclasses of A, it is safe to read the list and cast its elements to type A.

```
public void print(List<? extends Animal> myList){  
    for(Animal a: myList){  
        System.out.println(a);  
    }  
}  
  
List<Dog> dogs = new ArrayList<Dog>();  
print(dogs);  
List<Bear> bears = new ArrayList<Bear>();  
print(bears);
```

- This can be thought of as read-only collection

The Super Wildcard

- The `List<? super A>` means a list of objects that are instances of the class A, or superclasses of A.
- Since we do know the type of elements of the list is A or superclasses of A, it is safe to insert elements of type A or subclasses of A to the list.

```
public void insert(List<? super Animal> myList){  
    myList.add(new Dog());  
    myList.add(new Bear());  
}  
  
List<Animal> animals = new ArrayList<Animal>();  
insert(animals);  
Object o = animals.get(0); // OK  
Animal a = animals.get(0) // Not OK  
  
List<Object> objects = new ArrayList<Object>();  
insert(objects);
```

Subtyping - continue

- Upcasting and downcasting types:

```
abstract class Animal {}
class Dog extends Animal {}
class Bear extends Animal {}

Dog dog = new Dog();
Animal a = (Animal) dog; // upcasting Dog -> Animal

Animal b = new Dog();
Dog d = (Dog) b; // downcasting Animal -> Dog

Animal c = new Animal(){...};
Dog e = (Dog) c; // Runtime error: ClassCastException
                 // c could be a bear, which can't
                 // be cast to a dog.
```

Subtyping - continue

- In general, $T1<X> <: T2<X>$ if $T1 <: T2$, where $<:$ denotes *is subtype of*.
- Now let's consider:

```
ArrayList<String> arrayListStr = new ArrayList<String>();  
List<String> listStr = arrayListStr; // Is this OK?
```

- In this case, it's fine. Since `ArrayList` is subtype of `List`, we have `ArrayList<String>` as subtype of `List<String>`. Thus, "`List<String> listStr = arrayListStr`" is upcasting type from `ArrayList<String>` to `List<String>`.

Generic Methods

Keyword

Generic Method: A method that accepts arguments, or returns objects, of an arbitrary type.

A generic method can be defined in any class. The type parameter (e.g. T) is *local* to the method.

```
public <T> int genericMethod(T arg); // Generic argument  
  
public <T> T genericMethod(String name); // Generic return value  
  
public <T> T genericMethod(T arg); // Both!  
  
public <T,S> T genericMethod(S arg); // Both!
```

Assess Yourself

Write a generic method that accepts two arguments:

- array: an array of elements whose type is T
- item: an object of the same type as the array's elements

The method should return a count of how many times item appears in array.

Assess Yourself

```
public class TestGenericMethods {  
    public static void main(String[] args) {  
        Integer[] nums = {1, 3, 6, 9, 3, 5, 9, 3, 5, 42, null};  
        String[] names = {"Jon", "Arya", "Dany", "Tyrion", "Jon"};  
        System.out.println(countOccurrences(nums, 3));  
        System.out.println(countOccurrences(names, "Jon"));  
    }  
  
    public static <T> int countOccurrences(T[] array, T item) {  
        int count = 0;  
        if (item == null) {  
            for (T arrayItem : array){  
                count = arrayItem == item ? count + 1 : count;  
            }  
        } else {  
            for (T arrayItem : array){  
                count = item.equals(arrayItem) ? count + 1 : count;  
            }  
        }  
        return count;  
    }  
}
```

Pitfall: What Can't We Do?

Generic programming is powerful, but has its limitations. When using generics, we can't:

- Instantiate parametrized objects

```
T item = new T();
```

- Create arrays of parametrized objects

```
T[] elements = new T[];
```

Otherwise, most things are fair game.

Learning Outcomes

You should be able to:

- Understand **generic** classes in Java
- Use **generically typed** classes
- Define **generically typed** classes
- Understand generics' subtyping

SWEN20003
Object Oriented Software Development

Collections and Maps

Semester 1, 2002

Lecture Objectives

After this lecture you will be able to:

- Choose appropriate data structures storing, retrieving and manipulating objects (data)
- Use the Java Collections Framework
- Use the Java Maps Framework

Collections and Maps

Understanding how to store data (a collection of objects) for later retrieval and manipulation is an essential when writing programs.

Java provides two *frameworks* to support this.

Keyword

Collections: A framework that permits storing, accessing and manipulating *lists* (an ordered collection).

Keyword

Maps: A framework that permits storing, accessing and manipulating *key-value pairs*.

Back to ArrayList

Last lecture we looked at the `ArrayList` as a generic class.

`ArrayList` is a class in the Java Collections framework that can be used for storing, retrieving and manipulating a group of objects.

In this lecture we will take a closer look at how we can use the `ArrayList` class for more sophisticated data manipulations.

Using the ArrayList Class for storing

```
import java.util.ArrayList;
public class PrintCircleRadius {
    public static void main(String[] args) {
        ArrayList<Circle> circles = new ArrayList<Circle>();
        circles.add(new Circle(0.0, 0.0, 5));
        circles.add(new Circle(0.0, 0.0, 10));
        circles.add(new Circle(0.0, 0.0, 7));
        printRadius(circles);
    }
    private static void printRadius(ArrayList<Circle> circles){
        int index = 0;
        for(Circle c: circles) {
            System.out.println("Radius at index " + index +
                               " = " + c.getRadius());
            index++;
        }
    }
}
```

Using the ArrayList Class for sorting

Elements of an ArrayList can be easily sorted if:

The stored element class implements the Comparable<T> interface!

The compareTo() method of the class must provide a comparison (returning an integer) which will be used to decide how the elements are sorted.

Using the ArrayList Class for sorting

```
import java.util.*;
class Movie implements Comparable<Movie>
{
    private double rating;
    private String name;
    private int year;
    public Movie(String name, double rating, int year)
    {
        this.name = name;
        this.rating = rating;
        this.year = year;
    }
    public int compareTo(Movie m)
    {
        return this.year - m.year;
    }
    // Getters and setters go here - not shown
}
```

Using the ArrayList Class for sorting

```
import java.util.ArrayList;
import java.util.Collections;
public class MovieSorter {
    public static void main(String[] args) {
        ArrayList<Movie> list = new ArrayList<Movie>();
        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));
        list.add(new Movie("Return of the Jedi", 8.4, 1983));
        Collections.sort(list);
        printList(list);
    }

    public static void printList(ArrayList<Movie> list) {
        for (Movie movie: list)
            System.out.println(movie.getRating() + " " +
                               movie.getName() + " " + movie.getYear());
    }
}
```

Using the ArrayList Class for sorting

What would the program print?

- 8.7 Star Wars 1977
- 8.8 Empire Strikes Back 1980
- 8.4 Return of the Jedi 1983
- 8.3 Force Awakens 2015

Now, what if we want to sort the movies by rating or name - not year?

How can we do that?

Good news is java Comparator and Collections.sort() can still help you!

Using the ArrayList Class for sorting

```
import java.util.Comparator;
class RatingComparator implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        if (m1.getRating() < m2.getRating()) return -1;
        if (m1.getRating() > m2.getRating()) return 1;
        else return 0;
    }
}

import java.util.Comparator;
public class NameComparator implements Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {
        return m1.getName().compareTo(m2.getName());
    }
}
```

Using the ArrayList Class for sorting

```
// import statements
public class MovieSorter {
    public static void main(String[] args) {
        // Code to add movies to the arraylist - same as previous example
        Collections.sort(list);
        printList(list);
        System.out.println("*****");
        Collections.sort(list,new RatingComparator());
        printList(list);
        System.out.println("*****");
        Collections.sort(list,new NameComparator());
        printList(list);
    }
    public static void printList(ArrayList<Movie> list) {
        for (Movie movie: list)
            System.out.println(movie.getRating() + " " +
                               movie.getName() + " " + movie.getYear());
    }
}
```

Using the ArrayList Class for sorting

What would the program print?

8.7 Star Wars 1977

8.8 Empire Strikes Back 1980

8.4 Return of the Jedi 1983

8.3 Force Awakens 2015

8.3 Force Awakens 2015

8.4 Return of the Jedi 1983

8.7 Star Wars 1977

8.8 Empire Strikes Back 1980

8.8 Empire Strikes Back 1980

8.3 Force Awakens 2015

8.4 Return of the Jedi 1983

8.7 Star Wars 1977

Using the ArrayList Class for sorting

In the previous example, we developed new comparator class for each comparison.

Was it necessary? Is that a bit of an overkill?

Is there a different solution?

Anonymous Inner Class is the solution.

Keyword

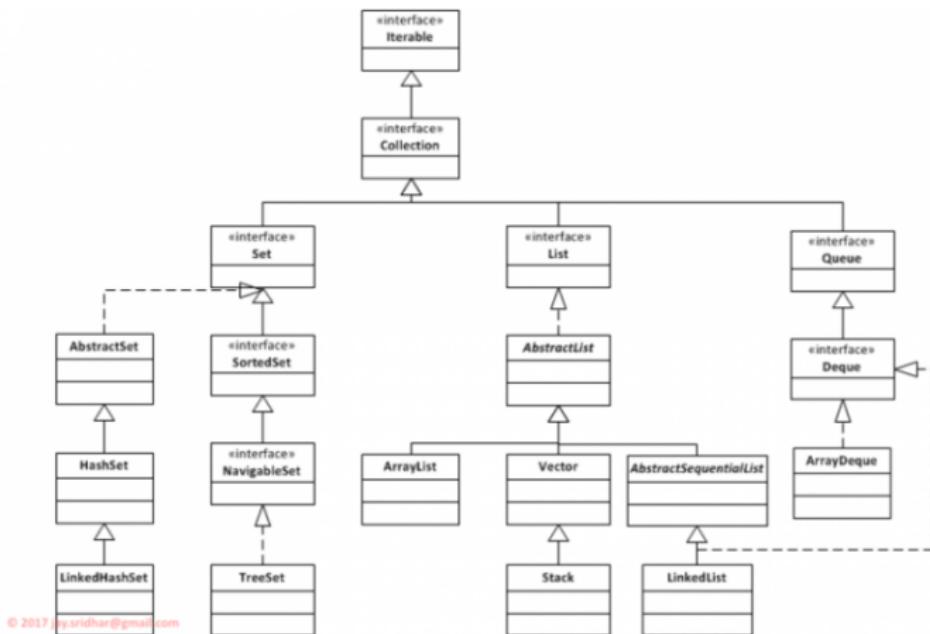
Anonymous Inner Class: A class created “on the fly”, without a new file, or class name for which only a single object is created.

Using the ArrayList Class for sorting

```
public class MovieSorterAnonymous {
    public static void main(String[] args) {
        // Same code as the previous example
        Collections.sort(list, new Comparator<Movie>(){
            @Override
            public int compare(Movie m1, Movie m2){
                if (m1.getRating() < m2.getRating()) return -1;
                if (m1.getRating() > m2.getRating()) return 1;
                else return 0;
            }});
        printList(list);

        Collections.sort(list, new Comparator<Movie>(){
            @Override
            public int compare(Movie m1, Movie m2) {
                return m1.getName().compareTo(m2.getName());
            }});
        printList(list);
    }
}
```

Collections Hierarchy



Java Collections Framework

Source: <https://dzone.com/articles/java-collections>

Common Operations - Collections

Length `int size()`

Presence `boolean contains(Object element)`

Only works when element defines
`equals(Object element)`

Add `boolean add(E element)`

Remove `boolean remove(Object element)`

Iterating `Iterator<E> iterator()`

Iterating `for (T t : Collection<T>)`

Retrieval `Object get(int index)`

Supported only at `AbstractList` level and below.

Most Useful?

Each of these have their useful applications, but personally...

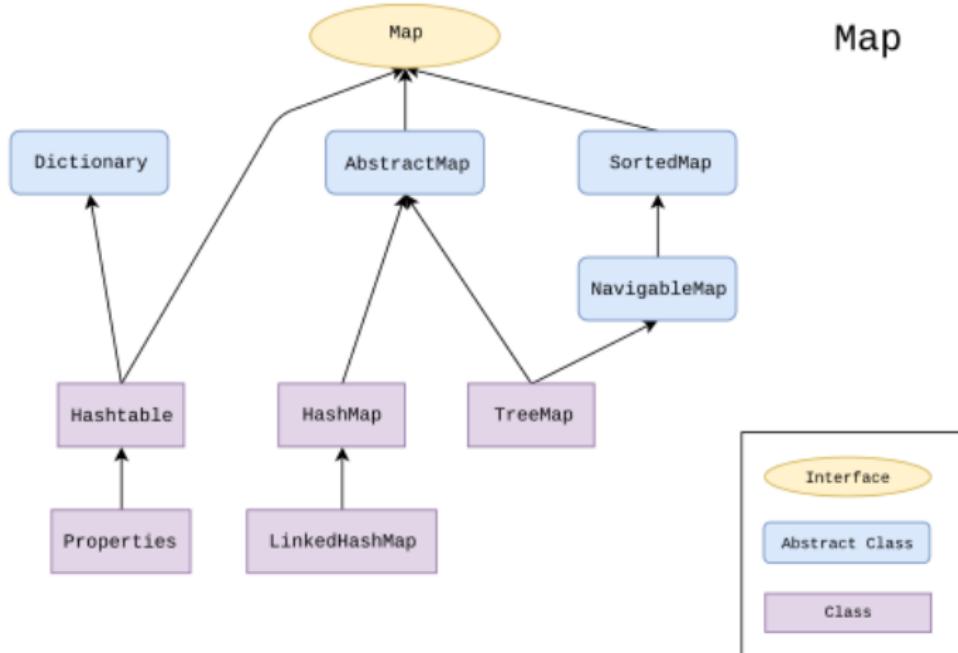
- `ArrayList`: like arrays, but better
- `HashSet`: ensures elements are unique - no duplicates
- `PriorityQueue`: allows you to *order* elements in non-trivial ways
- `TreeSet`: Fast lookup/search of unique elements

Maps

Keyword

Maps: A framework that permits storing, accessing and manipulating *key-value pairs*.

Maps Hierarchy



Source: https://en.wikipedia.org/wiki/Java_collections_framework [Note: Not UML]

Common Operations - Maps

Length `int size()`

Presence `boolean containKey(Object key)`

Presence `boolean containValue(Object value)`

Add/Replace `boolean put(K key, V value)`

Remove `boolean remove(Object key)`

Iterating `Set<K> keySet()`

Iterating `Set<Map.Entry<K,V>> entrySet()`

Retrieval `V get(Object key)`

Using HashMap

A generic class that takes two types: K (the key) and V (the value)

```
import java.util.HashMap;

public static void main(String[] args) {
    HashMap<String, Book> library = new HashMap<>();

    Book b1 = new Book("J.R.R. Tolkien", "The Lord of the Rings", 1178);
    Book b2 = new Book("George R. R. Martin", "A Game of Thrones", 694);

    library.put(b1.author, b1);
    library.put(b2.author, b2);

    for(String author : library.keySet()) {
        Book b = library.get(author);
        System.out.format("%s, %s, %d\n", b.getAuthor(),
                          b.getTitle(), b.getNumPages());
    }
}
```

Assess Yourself

If you were to create a digital phonebook using a HashMap, what would the key and value types be?

```
HashMap<String, Integer> phonebook = new HashMap<>();
```

Assess Yourself

If you were to create a system to link a pet's ID to it's owner, what would the key and value types be?

```
HashMap<Integer,Person> petTracker = new HashMap<>();
```

Assess Yourself

Write a class called Tracker, which accepts two type parameters. The first type must be subclass of Person, and the second type a subclass of Locator.

A Person object could be a Hiker, Diver, or Pilot.

A Locator object could be GPS, Infrared, or IP.

The Tracker class maintains a list of TwoTypePair objects, with the elements of the TwoTypePair being a Person and a Locator.

Generics in the Collections and Maps

If we didn't have generic classes, how would you implement a list, a map, etc.?

- Define everything as Object
- Rewrite your code for any type you might use it with

Generics give us **flexibility**; code once, reuse the code for **any** type. They also allow objects to keep their **type** (i.e. not be Objects), **and**, allows the compiler to detect errors, thereby prevent run-time errors if code is properly designed.

Lecture Objectives

After this lecture you will be able to:

- Choose appropriate data structures storing, retrieving and manipulating objects (data)
- Use Java Collections Framework
- Use Java Maps Framework

Assess Yourself

Explain (using examples) the difference between a *concrete* and *abstract* class.

Explain what the `super` and `this` keywords refer to.

Assess Yourself

Explain (**using examples**) the difference between a *concrete* and *abstract* class.

Assess Yourself

Explain (**using examples**) the difference between a *concrete* and *abstract* class.

Abstract classes are missing some information and can't be instantiated. (**1 mark**)

Assess Yourself

Explain (**using examples**) the difference between a *concrete* and *abstract* class.

Abstract classes are missing some information and can't be instantiated. (**1 mark**)

Concrete classes are fully complete, and have all the information they need. (**1 mark**)

Assess Yourself

Explain (**using examples**) the difference between a *concrete* and *abstract* class.

Abstract classes are missing some information and can't be instantiated. (**1 mark**)

Concrete classes are fully complete, and have all the information they need. (**1 mark**)

For example: a Vehicle may have wings or wheels, so it is abstract, but a Car is concrete. (**1 mark**)

Assess Yourself

Explain what the `super` and `this` keywords refer to.

`super` refers to the calling object's parent class. **(1 mark)**

Assess Yourself

Explain what the `super` and `this` keywords refer to.

`super` refers to the calling object's parent class. **(1 mark)**

`this` refers to the calling object itself. **(1 mark)**

SWEN20003
Object Oriented Software Development

Exceptions

Semester 2, 2019

The Road So Far

- Java Foundations
- Classes and Objects
 - ▶ Encapsulation
 - ▶ Information Hiding (Privacy)
- Inheritance and Polymorphism
 - ▶ Inheritance
 - ▶ Polymorphism
 - ▶ Abstract Classes
 - ▶ Interfaces
- Modelling classes and relationships

Lecture Objectives

After this lecture you will be able to:

- Understand what **exceptions** are
- Appropriately handle **exceptions** in Java
- Define and utilise **exceptions** in Java

Errors

It is common to make mistakes (errors) while developing as well as typing a program.

Such mistakes can be categorised as:

- Syntax errors
- Semantic errors
- Runtime errors

Errors

Keyword

Syntax: Errors where what you write isn't legal code; identified by the editor/compiler.

Errors

Keyword

Syntax: Errors where what you write isn't legal code; identified by the editor/compiler.

Keyword

Semantic: Code runs to completion, but results in *incorrect* output/operation; identified through software testing (coming soon).

Errors

Keyword

Syntax: Errors where what you write isn't legal code; identified by the editor/compiler.

Keyword

Semantic: Code runs to completion, but results in *incorrect* output/operation; identified through software testing (coming soon).

Keyword

Runtime: An error that causes your program to end prematurely (crash and burn); identified through execution.

Common Runtime Errors

- Dividing a number by zero
- Accessing an element that is out of bounds of an array.
- Trying to store incompatible data elements.
- Using negative value as array size
- Trying to convert from string data to another type (e.g., converting string “abc” to integer value)
- File errors:
 - ▶ opening a file in “read mode” that does not exist or no read permission
 - ▶ Opening a file in “write/update mode” which has “read only” permission
- Many more ...

Runtime Error - Example

```
class NoErrorHandler {
    public static void main(String[] args){
        int n1 = 1, n2 = 0;
        System.out.println("The result is " + divide(n1, n2));
        System.out.println("The program reached this line");
    }
    public static int divide(int n1, int n2) {
        return n1/n2;
    }
}
```

What happens if $n2 == 0$?

Runtime Error - Example

```
class NoErrorHandler {
    public static void main(String[] args){
        int n1 = 1, n2 = 0;
        System.out.println("The result is " + divide(n1, n2));
        System.out.println("The program reached this line");
    }
    public static int divide(int n1, int n2) {
        return n1/n2;
    }
}
```

What happens if $n2 == 0$?

Exception in thread "main" java.lang.ArithmetiException: ...

Runtime Error - Example

```
class NoErrorHandler {
    public static void main(String[] args){
        int n1 = 1, n2 = 0;
        System.out.println("The result is " + divide(n1, n2));
        System.out.println("The program reached this line");
    }
    public static int divide(int n1, int n2) {
        return n1/n2;
    }
}
```

What happens if $n2 == 0$?

Exception in thread "main" java.lang.ArithmetiException: ...

Solution 1: Do nothing and hope for the best.
Obviously less than ideal.

Runtime Errors

How can we protect against the error?

Runtime Errors

How can we protect against the error?

```
public int divide(int n1, double n2) {  
    if (n2 != 0) {  
        return n1/n2;  
    } else {  
        ???  
    }  
}
```

Runtime Errors

How can we protect against the error?

```
public int divide(int n1, double n2) {  
    if (n2 != 0) {  
        return n1/n2;  
    } else {  
        ???  
    }  
}
```

```
if (n2 != 0) {  
    divide(n1, n2);  
} else {  
    // Print error message and exit or continue  
}
```

Runtime Errors

How can we protect against the error?

```
public int divide(int n1, double n2) {  
    if (n2 != 0) {  
        return n1/n2;  
    } else {  
        ???  
    }  
}
```

```
if (n2 != 0) {  
    divide(n1, n2);  
} else {  
    // Print error message and exit or continue  
}
```

Solution 2: Explicitly guard yourself against dangerous or invalid conditions, known as *defensive programming*.

Runtime Errors

What are some downsides of solution 2?

Runtime Errors

What are some downsides of solution 2?

- Need to explicitly protect against every possible error condition
- Some conditions don't have a "backup" or alternate path, they're just failures
- Not very nice to read
- Poor abstraction (bloated code)

Runtime Errors

```
class WithExceptionHandling {
    public static void main(String[] args){
        int n1 = 1, n2 = 0;

        try {
            System.out.println("The result is " + divide(n1, n2));
        } catch (ArithmetcException e) {
            System.out.println("Cannot divide - n2 is zero");
        }
        System.out.println("The program reached this line");
    }

    public static int divide(int n1, int n2) {
        return n1/n2;
    }
}
```

Runtime Errors

```
class WithExceptionHandling {
    public static void main(String[] args){
        int n1 = 1, n2 = 0;

        try {
            System.out.println("The result is " + divide(n1, n2));
        } catch (ArithméticaException e) {
            System.out.println("Cannot divide - n2 is zero");
        }
        System.out.println("The program reached this line");
    }

    public static int divide(int n1, int n2) {
        return n1/n2;
    }
}
```

Solution 3: Use exceptions to catch error states, then recover from them, or gracefully end the program.

Exceptions

Keyword

Exception: An *error state* created by a *runtime error* in your code; an exception.

Exceptions

Keyword

Exception: An *error state* created by a *runtime error* in your code; an exception.

Keyword

Exception: An object created by Java to *represent* the error that was encountered.

Exceptions

Keyword

Exception: An *error state* created by a *runtime error* in your code; an exception.

Keyword

Exception: An object created by Java to *represent* the error that was encountered.

Keyword

Exception Handling: Code that actively protects your program in the case of exceptions.

Exception Handling

```
public void method(...) {  
    try {  
        <block of code to execute,  
                     which may cause an exception>  
    } catch (<ExceptionClass> varName) {  
        <block of code to execute to recover from exception,  
                     or end the program>  
    } finally {  
        <block of code that executes whether an exception  
                     happened or not>  
    }  
}
```

Exception Handling

Keyword

try: Attempt to execute some code that may result in an error state (exception).

Exception Handling

Keyword

try: Attempt to execute some code that may result in an error state (exception).

Keyword

catch: Deal with the exception. This could be recovery (ask the user to input again, adjust an index) or failure (output an error message and exit).

Exception Handling

Keyword

try: Attempt to execute some code that may result in an error state (exception).

Keyword

catch: Deal with the exception. This could be recovery (ask the user to input again, adjust an index) or failure (output an error message and exit).

Keyword

finally: Perform clean up (like closing files) assuming the code didn't exit.

Exception Handling

```
class WithExceptionCatchThrowFinally {
    public static void main(String[] args){
        int n1 = 1, n2 = 0;

        try {
            System.out.println("The result is " + divide(n1, n2));
        } catch (ArithmetricException e) {
            System.out.println("Cannot divide - n2 is zero");
        } finally {
            System.out.println("The program reached this line");
        }
    }

    public static int divide(int n1, int n2) {
        return n1/n2;
    }
}
```

Exception Handling - Chaining Exceptions

```
public void processFile(String filename) {  
    try {  
        ...  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

We can also *chain* catch blocks to deal with different exceptions *separately*. The most “specific” exception (subclasses) come first, with “broader” exceptions (superclasses) listed lower.

Assess Yourself

Write a method that has the potential to create an `ArithmeticException` and an `ArrayIndexOutOfBoundsException`, and implement appropriate exception handling for these cases.

Assess Yourself

```
public class AverageDifference {
    public static void main(String[] args) {
        int[] n1 = {1, 2, 3};
        int[] n2 = {2, 3, 4};

        try {
            System.out.println("Answer = " + averageDifference(n1, n2));
        } catch (ArithmaticException e) {
            System.out.println("Caught an arithmetic exception");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught an index exception");
        }
    }

    public static int averageDifference(int n1[], int n2[]) {
        int sumDifference = 0;
        for (int i = 0; i < n1.length; i++) {
            sumDifference += n1[i] - n2[i];
        }
        return sumDifference/n1.length;
    }
}
```

Generating Exceptions

Keyword

throw: Respond to an error state by creating an exception object, either already existing or one defined by you.

Keyword

throws: Indicates a method has the potential to create an exception, and can't be bothered to deal with it (Slick stupidly does this for **everything**), or that the exact response varies by application.

Assess Yourself

Write a method that has the potential to create a NullPointerException, and throws an exception if its argument is null.

Assess Yourself

Write a method that has the potential to create a NullPointerException, and throws an exception if its argument is null.

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(int age, String name) {  
        if (name == null) {  
            throw new NullPointerException("Creating person with null name");  
        }  
        this.age = age;  
        this.name = name;  
    }  
  
    public static void main(String[] args) {  
        Person p1 = new Person(10, "Sarah");  
        Person p2 = new Person(12, null);  
    }  
}
```

Defining Exceptions

What if we discover a new “type” of problem?

Defining Exceptions

What if we discover a new “type” of problem?

We can define our own exceptions!

Defining Exceptions

What if we discover a new “type” of problem?

We can define our own exceptions!

- Exceptions are classes!
- Most exceptions inherit from an Exception class
- All exceptions should have two constructors, but we can add whatever else we like

Assess Yourself

Write a class Circle, which has attributes centre and radius, initialized at creation. Your must ensure that the radius is greater than zero.

Assess Yourself

Step 1: Write the exception class.

```
import java.lang.Exception;

public class InvalidRadiusException extends Exception {
    public InvalidRadiusException() {
        super("Radius is not valid");
    }

    public InvalidRadiusException(double radius){
        super("Radius [" + radius + "] is not valid");
    }
}
```

Assess Yourself

Step 2: Write the Circle class.

```
public class Circle {  
    private double centreX, centreY;  
    private double radius;  
  
    public Circle (double centreX, double centreY, double radius)  
        throws InvalidRadiusException {  
        if (r <= 0 ) {  
            throw new InvalidRadiusException(radius);  
        }  
        this.centreX = centreX;  
        this.centreY = centreY;  
        this.radius = radius;  
    }  
}
```

Assess Yourself

Step 3: Test your class.

```
public class TestCircle {  
    public static void main(String[] args) {  
        try {  
            Circle c1 = new Circle(10, 10, 100);  
            System.out.println("Circle 1 created");  
  
            Circle c2 = new Circle(10, 10, -1);  
            System.out.println("Circle 2 created");  
        } catch(InvalidRadiusException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

"Circle 1 created"

"Radius [-1] is not valid"

Types of Exceptions

Keyword

Unchecked: Inherit from the Error class. Can be safely ignored by the programmer; most (inbuilt) Java exceptions are *unchecked*, because you aren't forced to protect against them.

Keyword

Checked: Inherit from the Exception class. Must be explicitly handled by the programmer in some way; the compiler gives an error if a checked exception is ignored.

Exception Handling

Catch or Declare

- All checked exceptions must be handled by
 - ▶ Enclosing code that can generate exceptions in a try-catch block
 - ▶ Declaring that a method may create an exception using the `throws` clause
- Both techniques can be used in the same method, for different exceptions

Using Exceptions

- Should be reserved for when a method encounters an *unusual* or *unexpected* case that cannot be handled easily in some other way

Try With

```
public void processFile(String filename) {  
    try (BufferedReader reader = ...) {  
        ...  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

The **finally** block is used to handle cleanup, but in many cases we can use the **try-with** notation to handle cleanup automatically.

Questions?

Mid-semester Test Feedback

SWEN20003
Object Oriented Software Development

Design Patterns

Semester 1, 2020

The Road So Far

- Java Foundations
- Object Oriented Principles
 - ▶ Encapsulation
 - ▶ Information Hiding (Privacy)
 - ▶ Inheritance and Polymorphism
 - ▶ Abstract Classes
 - ▶ Interfaces
- Modelling classes and relationships
- Advanced Java
 - ▶ Generics
 - ▶ Exceptions

Lecture Objectives

After this lecture you will be able to:

- Describe what **design patterns** are, and why they are useful
- Analyse and understand **design pattern specifications**
- Make use of (some) design patterns

Software Design

Are you already a good software designer?

Software Design

Are you already a good software designer?

If not, how do you become a good software designer?

Software Design

Are you already a good software designer?

If not, how do you become a good software designer?

Follow three simple steps...

How to become a good software designer

Step 1: Learn the fundamentals of programming

- Programming Languages
- Algorithms and Data Structures

How to become a good software designer

Step 1: Learn the fundamentals of programming

- Programming Languages
- Algorithms and Data Structures

Step 2: Learn design paradigms and principles

- Structured Design
- Object Oriented Design

How to become a good software designer

Step 1: Learn the fundamentals of programming

- Programming Languages
- Algorithms and Data Structures

Step 2: Learn design paradigms and principles

- Structured Design
- Object Oriented Design

Step 3: Study and mimic the designs of experienced designers

- Good designers reuse solutions - they do not design everything from scratch
- **Design Patterns** systematically document re-occurring design solutions so that they can be reused

Design Patterns

A Software *Design Pattern* is a description of a solution to a recurring problem in software design.

The recurring nature of the problems makes the solution useful to software developers.

Publishing it in the form of a Pattern enables the solution to be used without reinventing the wheel.

Pioneering work on design patterns was done by Eric Gamma and the team who authored the classic book on design patterns ('Gang of Four'):

Analysing and Publishing a Pattern

Analysing and Publishing a Pattern

Intent The goal of the pattern, why it exists

Analysing and Publishing a Pattern

Intent The goal of the pattern, why it exists

Motivation A scenario that highlights a need for the pattern

Analysing and Publishing a Pattern

Intent The goal of the pattern, why it exists

Motivation A scenario that highlights a need for the pattern

Applicability General situations where you can use the pattern

Analysing and Publishing a Pattern

Intent The goal of the pattern, why it exists

Motivation A scenario that highlights a need for the pattern

Applicability General situations where you can use the pattern

Structure Graphical representations of the pattern, likely a UML class diagram

Analysing and Publishing a Pattern

Intent The goal of the pattern, why it exists

Motivation A scenario that highlights a need for the pattern

Applicability General situations where you can use the pattern

Structure Graphical representations of the pattern, likely a UML class diagram

Participants List of classes/objects and their roles in the pattern

Analysing and Publishing a Pattern

Intent The goal of the pattern, why it exists

Motivation A scenario that highlights a need for the pattern

Applicability General situations where you can use the pattern

Structure Graphical representations of the pattern, likely a UML class diagram

Participants List of classes/objects and their roles in the pattern

Collaboration How the objects in the pattern interact

Analysing and Publishing a Pattern

- Intent The goal of the pattern, why it exists
- Motivation A scenario that highlights a need for the pattern
- Applicability General situations where you can use the pattern
- Structure Graphical representations of the pattern, likely a UML class diagram
- Participants List of classes/objects and their roles in the pattern
- Collaboration How the objects in the pattern interact
- Consequences A description of the results, side effects, and tradeoffs when using the pattern

Analysing and Publishing a Pattern

Intent The goal of the pattern, why it exists

Motivation A scenario that highlights a need for the pattern

Applicability General situations where you can use the pattern

Structure Graphical representations of the pattern, likely a UML class diagram

Participants List of classes/objects and their roles in the pattern

Collaboration How the objects in the pattern interact

Consequences A description of the results, side effects, and tradeoffs when using the pattern

Implementation Example of “solving a problem” with the pattern

Analysing and Publishing a Pattern

Intent The goal of the pattern, why it exists

Motivation A scenario that highlights a need for the pattern

Applicability General situations where you can use the pattern

Structure Graphical representations of the pattern, likely a UML class diagram

Participants List of classes/objects and their roles in the pattern

Collaboration How the objects in the pattern interact

Consequences A description of the results, side effects, and tradeoffs when using the pattern

Implementation Example of “solving a problem” with the pattern

Known Uses Specific, real-world examples of using the pattern

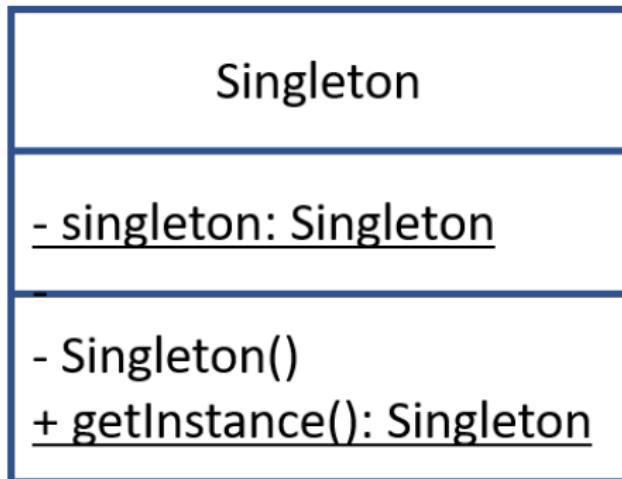
Design Patterns

Thousands of design patterns exist today.

Let us take a look as some common design patterns...

Singleton Pattern

Ensure that a class has only one instance and provide a global point of access to it.



Singleton Pattern - Implementation

```
class Singleton {  
    private static Singleton _instance = null;  
    private Singleton() {  
        //fill in the blank  
    }  
  
    public static Singleton getInstance() {  
        if ( _instance == null )  
            _instance = new Singleton();  
        return _instance;  
    }  
  
    public void otherOperations() { }  
}
```

Singleton Pattern - Collaboration

```
class TestSingleton {  
  
    public void method1(){  
        X = Singleton.getInstance();  
    }  
  
    public void method2(){  
        Y = Singleton.getInstance();  
    }  
}
```

Singleton Pattern Analysis

Singleton Pattern Analysis

Intent Ensure that a class has only one instance and provide a global point of access to it.

Singleton Pattern Analysis

Intent Ensure that a class has only one instance and provide a global point of access to it.

Motivation There are cases where only one instance of a class must be enforced with easy access to the object.

Singleton Pattern Analysis

Intent Ensure that a class has only one instance and provide a global point of access to it.

Motivation There are cases where only one instance of a class must be enforced with easy access to the object.

Applicability Use when a single instance of a class is required.

Structure See previous slides

Singleton Pattern Analysis

Intent Ensure that a class has only one instance and provide a global point of access to it.

Motivation There are cases where only one instance of a class must be enforced with easy access to the object.

Applicability Use when a single instance of a class is required.

Structure See previous slides

Participants Singleton class.

Singleton Pattern Analysis

Intent Ensure that a class has only one instance and provide a global point of access to it.

Motivation There are cases where only one instance of a class must be enforced with easy access to the object.

Applicability Use when a single instance of a class is required.

Structure See previous slides

Participants Singleton class.

Collaboration See previous slides.

Singleton Pattern Analysis

Intent Ensure that a class has only one instance and provide a global point of access to it.

Motivation There are cases where only one instance of a class must be enforced with easy access to the object.

Applicability Use when a single instance of a class is required.

Structure See previous slides

Participants Singleton class.

Collaboration See previous slides.

Consequences Use it with caution because inappropriate use could result in a bad design.

Singleton Pattern Analysis

Intent Ensure that a class has only one instance and provide a global point of access to it.

Motivation There are cases where only one instance of a class must be enforced with easy access to the object.

Applicability Use when a single instance of a class is required.

Structure See previous slides

Participants Singleton class.

Collaboration See previous slides.

Consequences Use it with caution because inappropriate use could result in a bad design.

Implementation See previous slides.

Singleton Pattern Analysis

Intent Ensure that a class has only one instance and provide a global point of access to it.

Motivation There are cases where only one instance of a class must be enforced with easy access to the object.

Applicability Use when a single instance of a class is required.

Structure See previous slides

Participants Singleton class.

Collaboration See previous slides.

Consequences Use it with caution because inappropriate use could result in a bad design.

Implementation See previous slides.

Known Uses e.g. CacheManager class, PrinterSpooler class.



Template Method and Strategy Patterns

Building generic components that can be *extended*,
adapted and *re-used* is key to good design.

Template Method and Strategy Patterns

Building generic components that can be *extended*, *adapted* and *re-used* is key to good design.

The two main techniques that are used for developing generic components are *refactoring* and *generalizing*.

- identifying recurring code and replacing with generic code

Template Method and Strategy Patterns

Building generic components that can be *extended*, *adapted* and *re-used* is key to good design.

The two main techniques that are used for developing generic components are *refactoring* and *generalizing*.

- identifying recurring code and replacing with generic code

Inheritance and *delegation* are the two main OO design techniques that are used for building generic components.

Template Method and Strategy Patterns

Template Method and *Strategy* are two design patterns that solve the problem of separating a generic algorithm from a detailed design.

Template Method pattern uses *Inheritance*.

Strategy pattern uses *Delegation*.

Template Method Pattern Motivation

```
public class BubbleSorter {  
    static int operations = 0;  
    public static int sort(int[] array){  
        operations = 0;  
        if (array.length <= 1)  
            return operations;  
        for (int i = array.length - 2; i >= 0; i --){  
            for (int j = 0; j <= i; j++){  
                compareAndSwap(array, j);  
            }  
        }  
        return operations;  
    }  
}
```

contd...

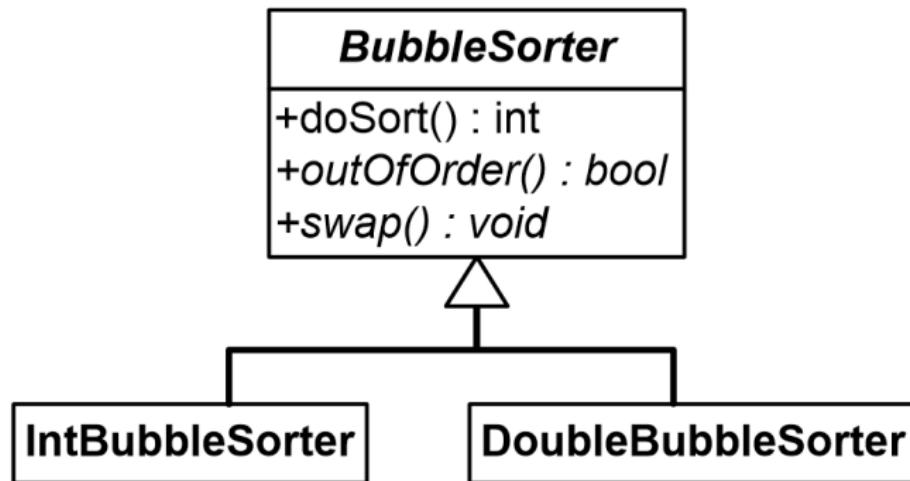
Template Method Pattern Motivation

```
public static void compareAndSwap(int[] array, int index){  
    if (array[index] > array[index+1])  
        swap(array, index);  
    operations++;  
}  
  
public static void swap(int[] array, int index){  
    int temp = array[index];  
    array[index] = array[index+1];  
    array[index+1] = temp;  
}  
}
```

Template Method Pattern Example

```
public abstract class AbstractBubbleSorter {  
    private static int operations = 0;  
    protected int length = 0;  
    protected int doSort(){  
        operations = 0;  
        if (length <= 1)  
            return operations;  
        for (int i = length - 2; i >= 0; i--){  
            for (int j = 0; j <= i; j++){  
                if (outOfOrder(j))  
                    swap(j);  
                operations++;  
            }  
        }  
        return operations;  
    }  
    protected abstract void swap(int index);  
    protected abstract boolean outOfOrder(int index);  
}
```

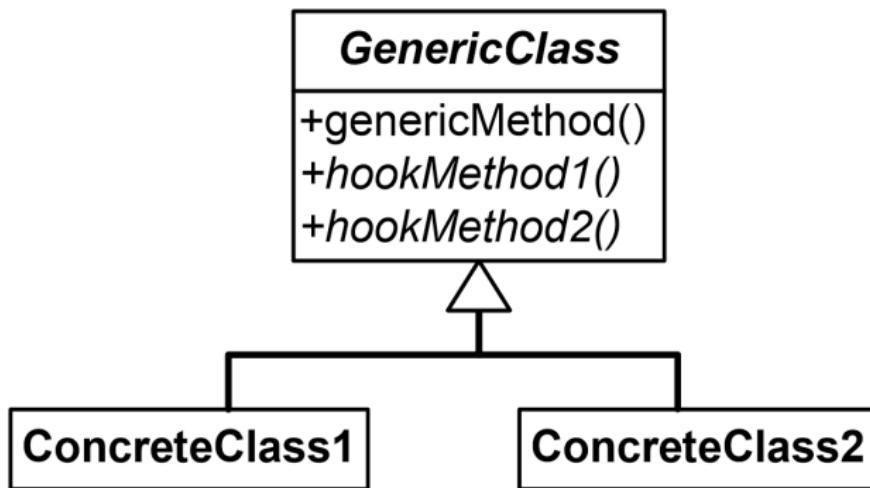
Template Method Pattern Example



Template Method Pattern Example

```
public class IntBubbleSorter extends AbstractBubbleSorter {  
    private int[] array = null;  
    public int sort(int[] a){  
        array = a;  
        length = array.length;  
        return doSort();  
    }  
    @Override  
    protected boolean outOfOrder(int index) {  
        return (array[index] > array[index+1]);  
    }  
    @Override  
    protected void swap(int index) {  
        int temp = array[index];  
        array[index] = array[index+1];  
        array[index+1] = temp;  
    }  
}
```

Template Method Pattern Structure



Template Method Pattern Analysis

Template Method Pattern Analysis

Intent Define a family of algorithms, encapsulate each one, and make them interchangeable.

Template Method Pattern Analysis

- Intent** Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Motivation** Build generic components that are easy to extend and reuse.

Template Method Pattern Analysis

- Intent** Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Motivation** Build generic components that are easy to extend and reuse.
- Applicability** Allows the implementation of invariant parts of an algorithm once and leave it to the subclass to implement the behavior that can vary.
- Structure** See previous slides.

Template Method Pattern Analysis

- Intent** Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Motivation** Build generic components that are easy to extend and reuse.
- Applicability** Allows the implementation of invariant parts of an algorithm once and leave it to the subclass to implement the behavior that can vary.
- Structure** See previous slides.
- Participants** See previous slides.

Template Method Pattern Analysis

- Intent** Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Motivation** Build generic components that are easy to extend and reuse.
- Applicability** Allows the implementation of invariant parts of an algorithm once and leave it to the subclass to implement the behavior that can vary.
- Structure** See previous slides.
- Participants** See previous slides.
- Collaboration** See previous slides.

Template Method Pattern Analysis

- Intent** Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Motivation** Build generic components that are easy to extend and reuse.
- Applicability** Allows the implementation of invariant parts of an algorithm once and leave it to the subclass to implement the behavior that can vary.
- Structure** See previous slides.
- Participants** See previous slides.
- Collaboration** See previous slides.
- Consequences** All algorithms must use the same interface.

Template Method Pattern Analysis

- Intent** Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Motivation** Build generic components that are easy to extend and reuse.
- Applicability** Allows the implementation of invariant parts of an algorithm once and leave it to the subclass to implement the behavior that can vary.
- Structure** See previous slides.
- Participants** See previous slides.
- Collaboration** See previous slides.
- Consequences** All algorithms must use the same interface.
- Implementation** See previous slides.

Template Method Pattern Analysis

- Intent** Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Motivation** Build generic components that are easy to extend and reuse.
- Applicability** Allows the implementation of invariant parts of an algorithm once and leave it to the subclass to implement the behavior that can vary.
- Structure** See previous slides.
- Participants** See previous slides.
- Collaboration** See previous slides.
- Consequences** All algorithms must use the same interface.
- Implementation** See previous slides.
- Known Uses** See previous slides.

Strategy Pattern

Template method is an example of using inheritance as a mechanism for re-use.

- Generic algorithm is placed in the base class
- Specific implementation is deferred to the sub class

The design tradeoff of using inheritance is the strong dependency to the base class.

- Although the methods `outOfOrder` and `swap` are generic methods they cannot be re-used because they inherit the `AbstractBubbleSorter` class

The Strategy pattern is an alternative.

Strategy Pattern Example

```
public class BubbleSorterS {  
  
    static int operations = 0;  
    private int length = 0;  
    private SortHandle itsSortHandle = null;  
  
    public BubbleSorterS(SortHandle handle){  
        itsSortHandle = handle;  
    }  
}
```

Contd..

Strategy Pattern Example

```
public int sort(Object array){  
    itsSortHandle.setArray(array);  
    length = itsSortHandle.length();  
    operations = 0;  
    if (length <= 1)  
        return operations;  
  
    for(int nextToLast = length - 2; nextToLast >=0;nextToLast-  
        for (int index=0; index <= nextToLast; index++){  
            if (itsSortHandle.outOfOrder(index))  
                itsSortHandle.swap(index);  
            operations++;  
        }  
    }  
    return operations;  
}
```

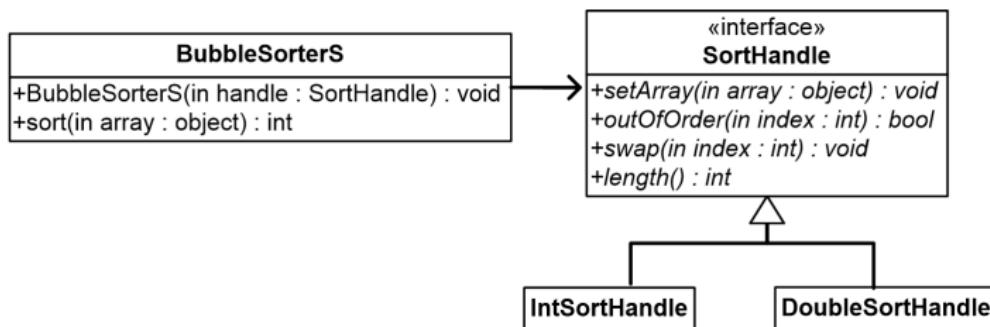
Strategy Pattern Example

```
public interface SortHandle {  
    public void swap(int index);  
    public boolean outOfOrder(int index);  
    public int length();  
    public void setArray(Object array);  
}
```

Strategy Pattern Example

```
public class IntSortHandle implements SortHandle {  
    private int[] array = null;  
  
    public int length() {  
        return array.length;  
    }  
    public boolean outOfOrder(int index) {  
        return (array[index] > array[index+1]);  
    }  
    public void setArray(Object array) {  
        this.array = (int [])array;  
    }  
    public void swap(int index) {  
        int temp = array[index];  
        array[index] = array[index + 1];  
        array[index + 1] = temp;  
    }  
}
```

Strategy Pattern Example



Strategy Pattern Example

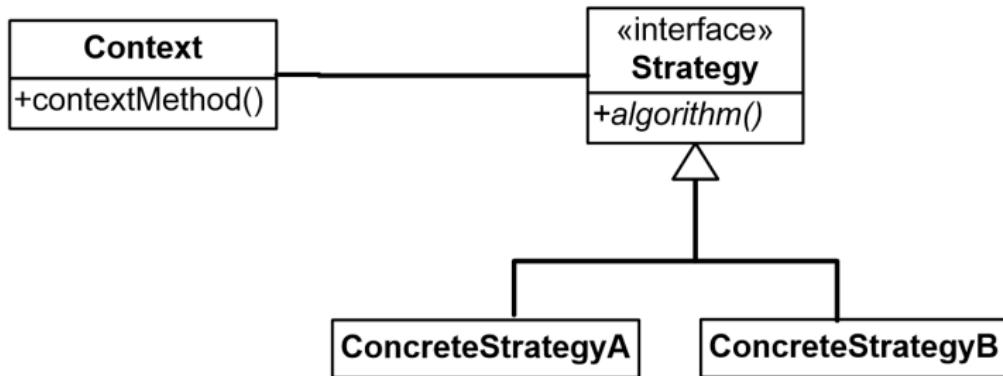
```
public class QuickBubbleSorterS {  
    static int operations = 0;  
    private int length = 0;  
    private SortHandle itsSortHandle = null;  
  
    public QuickBubbleSorterS(SortHandle handle){  
        itsSortHandle = handle;  
    }  
  
    public int sort(Object array){  
        itsSortHandle.setArray(array);  
        length = itsSortHandle.length();  
        operations = 0;  
    }  
}
```

Contd...

Strategy Pattern Analysis

```
if (length <= 1)
    return operations;
boolean thisPassInOrder = false;
for(int nextToLast = length - 2; nextToLast >=0 &&
    !thisPassInOrder; nextToLast--){
    thisPassInOrder = true;
    for (int index=0; index <= nextToLast; index++){
        if (itsSortHandle.outOfOrder(index)){
            itsSortHandle.swap(index);
            thisPassInOrder = false;
        }
        operations++;
    }
}
return operations;
}
```

Strategy Pattern Structure



Factory Method Pattern

Creating objects in the class that requires (uses) the objects is inflexible.

- It commits the class to a particular object
- Makes it impossible to change the instantiation without having to change the class.

Factory Method pattern solves this problem by:

- Defining a separate operation for creating an object.
- Creating an object by calling a factory method.

Factory Method Pattern

Keyword

Factory: A general technique for *manufacturing* (creating) objects.

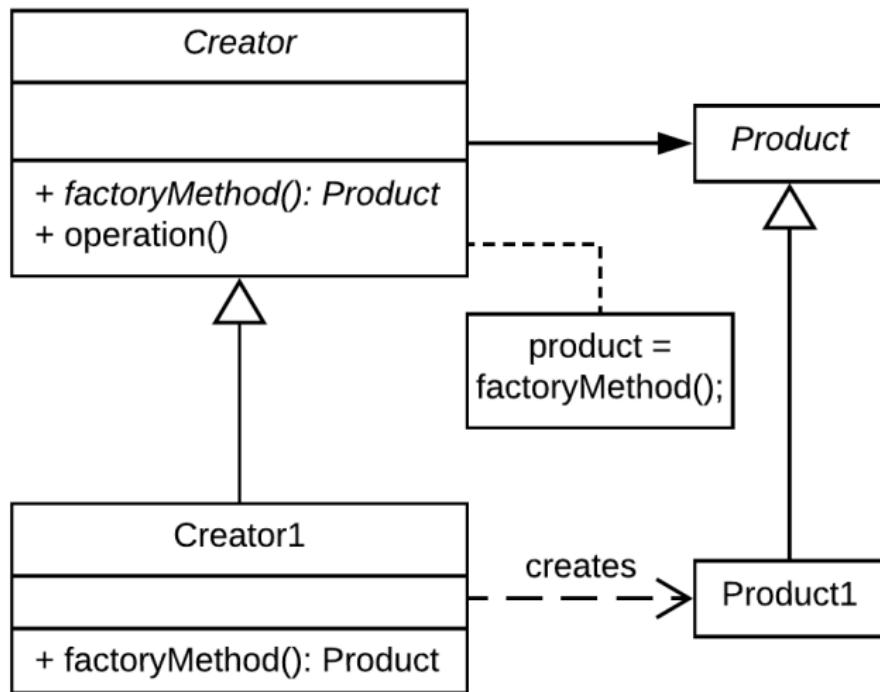
Keyword

Product: An abstract class that generalises the objects *being created/produced* by the factory.

Keyword

Creator: An abstract class that generalises the objects that *will consume/produce* products; generally have some *operation* (e.g. the constructor) that will invoke the factory method.

Factory Method Pattern Structure



Factory Method Pattern Motivation

But... What's wrong with how we'd normally do this?

Factory Method Pattern Motivation

But... What's wrong with how we'd normally do this?

- Can cause significant *code duplication*
- May require *inaccessible* information
- Not very well *abstracted*
- “Figuring out” which object to instantiate is not a primary concern for most classes
- Can make classes very *rigid* and *fragile*

Factory Method Pattern Motivation

But... What's wrong with how we'd normally do this?

- Can cause significant *code duplication*
- May require *inaccessible* information
- Not very well *abstracted*
- “Figuring out” which object to instantiate is not a primary concern for most classes
- Can make classes very *rigid* and *fragile*

Advantages of the Factory Method Pattern:

Factory Method Pattern Motivation

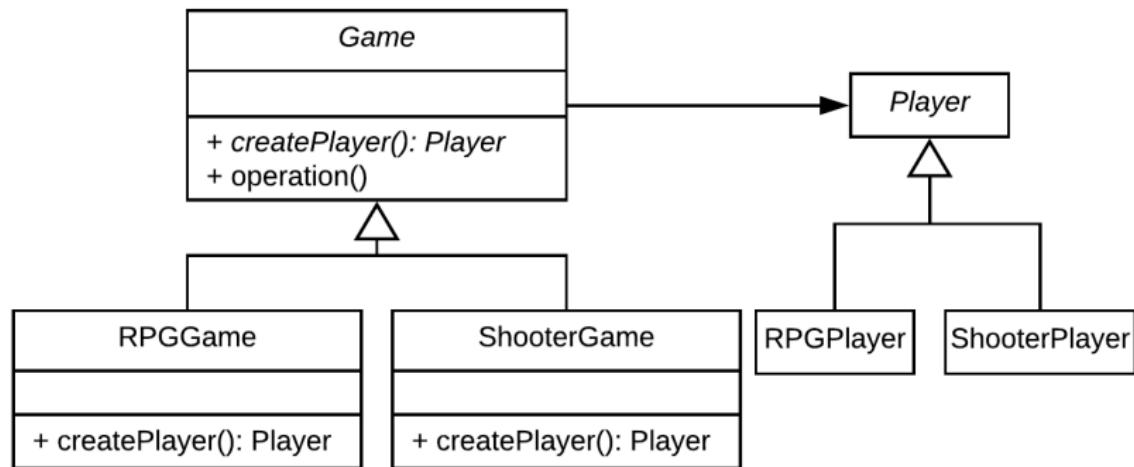
But... What's wrong with how we'd normally do this?

- Can cause significant *code duplication*
- May require *inaccessible* information
- Not very well *abstracted*
- “Figuring out” which object to instantiate is not a primary concern for most classes
- Can make classes very *rigid* and *fragile*

Advantages of the Factory Method Pattern:

- *Delegates* object creation (and the decision process) to subclasses
- *Abstracts* object creation by using a factory (object production) method
- *Encapsulates* objects by allowing *subclasses* to determine what they need

Factory Method Pattern Example



Factory Method Pattern Example

```
public abstract class Game {  
    private final List<Player> players = new ArrayList<>();  
  
    public Game(int nPlayers) {  
        for (int i = 0; i < nPlayers; i++) {  
            players.add(createPlayer());  
        }  
    }  
  
    public abstract Player createPlayer();  
}
```

Factory Method Patter Example

```
public class RPGGame extends Game {  
    @Override  
    public Player createPlayer() {  
        return new RPGPlayer();  
    }  
}  
  
public class ShooterGame extends Game {  
    @Override  
    public Player createPlayer() {  
        return new ShooterPlayer();  
    }  
}
```

```
RPGGame shadowQuest = new RPGGame();  
ShooterGame callOfDuty = new ShooterGame();
```

Factory Method Pattern Analysis

Factory Method Pattern Analysis

Intent To generalise object creation

Factory Method Pattern Analysis

Intent To generalise object creation

Motivation Loading player objects when a game loads

Factory Method Pattern Analysis

Intent To generalise object creation

Motivation Loading player objects when a game loads

Applicability When sister classes depend on (and create) similar objects

Factory Method Pattern Analysis

Intent To generalise object creation

Motivation Loading player objects when a game loads

Applicability When sister classes depend on (and create) similar objects

Structure See previous slides

Participants See previous slides

Collaboration Concrete creator objects invoke the factory method in order to produce their desired product

Consequences Object creation in the superclass is now *decoupled* from the specific object required

Factory Method Pattern Analysis

Intent To generalise object creation

Motivation Loading player objects when a game loads

Applicability When sister classes depend on (and create) similar objects

Structure See previous slides

Participants See previous slides

Collaboration Concrete creator objects invoke the factory method in order to produce their desired product

Consequences Object creation in the superclass is now *decoupled* from the specific object required

Implementation See previous slides

Known Uses See previous slides

Observer Pattern

There are situations where many objects (observers) depend on the state of one object (subject).

A single object managing a one-to-many dependency between objects is inflexible.

- It commits (tightly couples) the subject to particular dependent objects.
- Such tightly coupled objects are hard to implement, test and maintain.

Observer pattern decouples the subject and observers using a publish-subscribe style communication pattern.

Observer Pattern

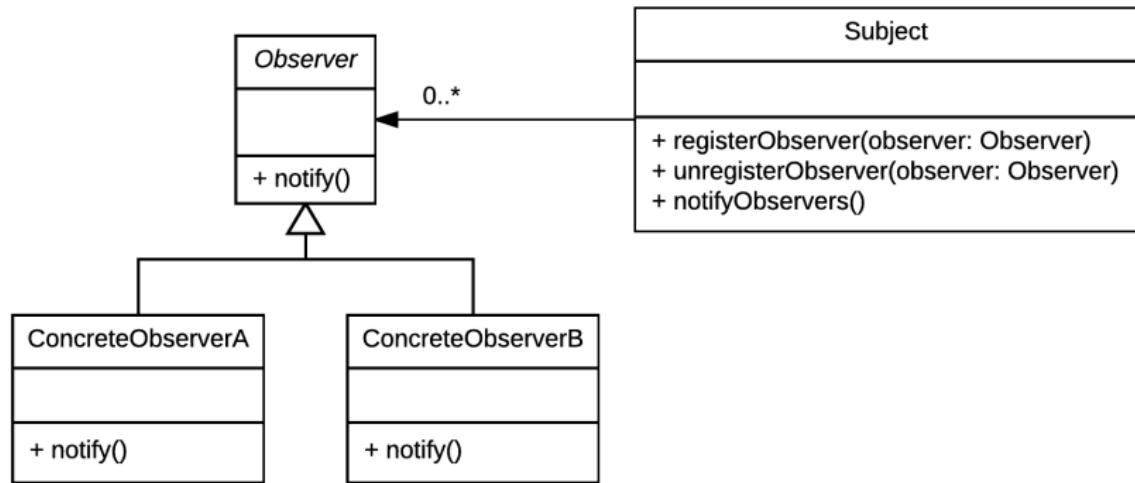
Keyword

Subject: An “important” object, whose state (or *change* in state) determines the actions of other classes.

Keyword

Observer: An object that monitors the subject in order to respond to its state, and any changes made to it.

Observer Pattern Structure



Observer Pattern

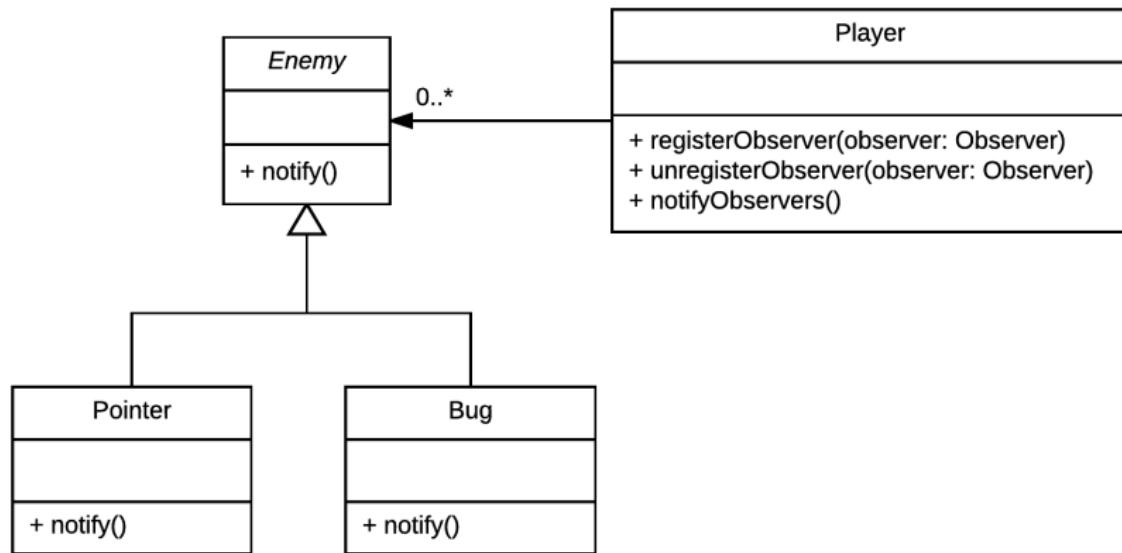
Why do we like this way better?

Observer Pattern

Why do we like this way better?

- Automatically notifies observers of *state changes*
- *Decouples* the subject and the observer
- With some extra tools, powers a great deal of *event-driven* programs
- Clear responsibilities: subjects (ideally) know nothing about the observers, except that they exist
- Observers can be added and removed from subjects at will, with zero effect (also known as the *publish-subscribe* model)
- Can be extended in various ways to improve messaging, decoupling, and event-handling

Observer Pattern Example



Note: this isn't actually a “Java-ready” design

Observer Pattern Example

```
import java.util.Observer;

public abstract class Enemy implements Observer {

    public abstract update(Observable o, Object arg);

}

public class Pointer extends Enemy {

    public update(Observable o, Object arg) {
        ...
    }
}

public class Bug extends Enemy {

    public update(Observable o, Object arg) {
        ...
    }
}
```

Observer Pattern Example

```
import java.util.Observable;

public class Player extends Observable {
    public Player(..., ArrayList<Observer> observers) {
        ...
        for (Observer o : observers) {
            this.addObserver(o);
        }
        notifyObservers("Player created");
    }
}
```

However, in most cases it makes more sense for Subject (or Observable) to be an interface, so we can write it ourselves.

Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software, a book written by the *Gang of Four* (or GoF) describing 23 common software design patterns.

All good developers have *seen* most of the patterns, and have an understanding of what they do, and what common problems have already been solved.

Don't reinvent the wheel... You'll learn more about patterns in SWEN30006.

Design Patterns

Commonly solved *classes* of problems:

Creational Solutions related to object creation,

e.g. Singleton, Factory Method

Structural Solutions dealing with the structure of

classes and their relationships,

e.g. Adapter, Bridge

Behavioural Solutions dealing with the interaction

among classes,

e.g. Strategy, Template Method, Observer

Review

- ① Identify where the patterns covered in the lecture can be applied to the project
- ② Describe the advantages of using design patterns (in general)
- ③ Describe the core components (the analysis) of the patterns covered in the lecture

Lecture Objectives

You should be able to:

- Describe what **design patterns** are, and why they are useful
- Analyse and understand **design pattern specifications**
- Make use of (some) design patterns

Assess Yourself

Explain what *problems* the Template and Strategy pattern attempt to solve, and how they are different.

Template and Strategy are two design patterns that solve the problem of separating a generic algorithm from a detailed design; situations where the *approach* is always the same, but the *implementation* can be different.

Template pattern uses Inheritance.

Strategy pattern uses Delegation.

SWEN20003
Object Oriented Software Development

Software Testing and Design

Semester 1, 2020

The Road So Far

- Java Foundations
- Classes and Objects
- Abstraction
- Advanced OOP
 - ▶ Exception Handling
 - ▶ Generic Classes
 - ▶ Generic Programming
 - ▶ Design Patterns
- Software Development Tools

Lecture Objectives

After this lecture you will be able to:

- **Write better** code
- **Design better** software
- **Test** your software for bugs

Documentation

Boring Stuff (Code Formatting)

While writing code is largely subjective, there are plenty of **conventions** that most programmers share:

- Use consistent layout (indentation, white space)
- Avoid long lines (80 characters is “historic”)
- Beware of tabs
- Lay out comments and code neatly
- Sensible naming of variables, method and classes
- Divide long files into sections with clear purposes
- Avoid copy and pasting/duplicating code
- Use a comment to explain each section

Boring Stuff (Comment Style)

While writing comments is largely subjective, there are plenty of **conventions** that most programmers share:

- Intended primarily for **yourself**, and developers writing code **with** you
- Code should be written to be **self-documenting**; readable without extra documentation
- Comments “tell the story” of the code
- If your code were removed, comments should be sufficient to “piece together” the algorithm
- Comments should be attached to *blocks* of code, which loosely correspond to *steps* in completing your algorithm

Boring Stuff (Comment Placement)

Bad Comment Placement

```
<line of code>
// This is a comment below my code
```

Terrible Comment Placement

```
<line of code> // This is an inline comment
```

Great Comment Placement

```
// This is a comment above my code
<line of code>
```

Comments appearing **before** code are like a “prologue” for your code; they introduce the *idea* of the code before you actually try to digest it.

Boring Stuff (Javadoc)

```
javadoc.equals(comments); // false
```

- Javadoc is a special kind of comment that can be **compiled to HTML**
- Intended primarily for developers **using** your program (exactly like Slick documentation)
- Used to document packages, classes, methods, and attributes (among others)
- Should document how to **use** and **interact** with your classes and their methods
- Various @ tags (like **@param** and **@return**) for generating specific documentation

Project Expectations (Javadoc)

You **must** include Javadoc documentation in your project 2 submission:

- **All** public classes, attributes, and methods
- Yes, this includes getters, setters, and constructors (hint, some things can be auto-generated)
- You **do not** need to use any fancy `@` tags, just provide `@param` and `@return` when appropriate
- No, we're not generating the HTML of your Javadoc

Software Design

Poor Design Symptoms

Think about how the following slide can be applied to your current/expected implementation of Project 2.

Imagine how difficult it would be for you to **change/fix/update** your solution if you identified a problem, or if the specification changed.

Poor Design Symptoms

Rigidity Hard to modify the system because changes in one class/method cascade to many others

Fragility Changing one part of the system causes unrelated parts to break

Immobility Cannot decompose the system into reusable modules

Viscosity Writing “hacks” when adding code in order to preserve the design

Complexity Lots of clever code that isn't necessary right now; premature optimisation is bad

Repetition Code looks like it was written by Cut and Paste

Opacity Lots of convoluted logic, design is hard to follow

GRASP

In SWEN30006 you'll learn about

- G** General
- R** Responsibility
- A** Assignment
- S** Software
- P** Patterns/Principles

Keyword

GRASP: A series of guidelines for assigning responsibility to classes in an object-oriented design; how to break a problem down into modules with clear purpose.

Keyword

Cohesion: Classes are designed to solve clear, focused problems. The class' methods/attributes are related to, and work towards, this objective. Designs should have **maximum** cohesion.

Keyword

Coupling: The degree of interaction between classes; dependency between classes. Designs should have **minimum** (low) coupling.

Keyword

Open-Closed Principle: Classes should be **open** to extension, but **closed** to modification.

In practice, this means if we need to *change* or *add* functionality to a class, we should not modify the original, but instead use **inheritance**.

Keyword

Abstraction: Solving problems by creating *abstract data types* to represent problem components; achieved in OOP through *classes*, which represent data and actions.

Keyword

Encapsulation: The details of a class should be kept *hidden* or *private*, and the user's ability to access the hidden details is *restricted* or *controlled*.
Also known as **data** or **information hiding**.

Keyword

Polymorphism: The ability to use an object or method in many different ways; achieved in Java through *ad hoc* (overloading), *subtype* (overriding, substitution), and *parametric* (generics) polymorphism.

Keyword

Delegation: Keeping classes *focused* by passing work to other classes. Computations should be performed in the class with the *greatest amount of relevant information*.

Software Testing

A bit about myself and my research



Keyword

Why software testing matters?: The cost of software bugs can be huge, e.g., peoples' lives and money.

Bug Fixing

How do you normally find/fix a bug?

- Print statements

```
System.out.println("Why does my code not reach here?");
```

- Google

How to fix my Java code

- Forums (Stackoverflow, etc.)

Someone please help my code is broken

Bug Fixing

Java offers a structured method for **testing**, very important for COMP30022:

Keyword

Unit: A small, well-defined component of a software system with one, or a small number, of responsibilities.

Keyword

Unit Test: Verifying the operation of a *unit* by testing a single *use case* (input/output), intending for it to **fail**.

Keyword

Unit Testing: Identifying bugs in software by subjecting every *unit* to a suite of *tests*.

Creating Units

What are the fundamental *units* of this method?

```
public boolean makeMove(Player player, Move move) {  
  
    int row = move.row;  
    int col = move.col;  
  
    if (row < 0 || row >= SIZE || col < 0 || col >= SIZE ||  
        !board[row][col].equals(EMPTY)) {  
        return false;  
    }  
  
    board[row][col] = player.getCharacter();  
  
    return true;  
}
```

Creating Units

```
public boolean cellIsEmpty(Move move) {  
    return board[move.row][move.col].equals(EMPTY);  
}
```

```
public boolean onBoard(Move move) {  
    return move.row >= 0 && move.row < SIZE &&  
        move.col >= 0 && move.col < SIZE;  
}
```

```
public boolean isValidMove(Move move) {  
    if (onBoard(move) && cellIsEmpty(move)) {  
        return true;  
    }  
    return false;  
}
```

```
public void makeMove(Player player, Move move) {  
    board[move.row][move.col] = player.getCharacter();  
}
```

Unit Testing With Java

Much better! What now?

Keyword

Manual Testing: Testing code manually, in an ad-hoc manner. Generally difficult to reach all edge cases, and not scalable for large projects.

Keyword

Automated Testing: Testing code with automated, purpose built software. Generally faster, more reliable, and less reliant on humans.

JUnit Testing

Keyword

assert: A true or false statement that indicates the success or failure of a test case.

Keyword

TestCase class: A class dedicated to testing a single unit.

Keyword

TestRunner class: A class dedicated to *executing* the tests on a unit.

TestCase Class

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BoardTest {
    @Test
    public void testBoard() {
        Board board = new Board();
        assertEquals(board.cellIsEmpty(0, 0), true);
    }

    @Test
    public void testValidMove() {
        Board board = new Board();
        Move move = new Move(0, 0);
        assertEquals(board.isValidMove(move), true);
    }
}
```

TestCase Class

```
@Test
public void testMakeMove() {
    Board board = new Board();
    Player player = new HumanPlayer("R");
    Move move = new Move(0, 0);
    board.makeMove(player, move);
    assertEquals(board.getBoard() [move.row] [move.col], "r");
}
```

TestRunner Class

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(BoardTest.class);

        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }

        System.out.println(result.wasSuccessful());
    }
}
```

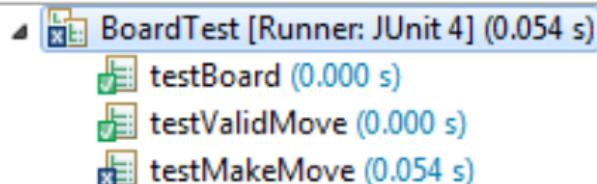
TestRunner Class

Finished after 0.075 seconds

Runs: 3/3

Errors: 0

Failures: 1



```
testMakeMove(BoardTest): expected:<[r]> but was:<[R]>  
false
```

Assess Yourself

Write a unit test to verify that when a move is made **off the board**, the `isValidMove` method returns false.

There are actually (at least) **four** test cases for this, but here's one:

```
@Test  
public void testValidMove2() {  
    Board board = new Board();  
    Move move = new Move(-1, 0);  
    assertEquals(false, board.isValidMove(move));  
}
```

JUnit Advantages

Large teams and open source development (**should**) **always** use automated testing:

- Easy to set up
- Scalable
- Repeatable
- Not human intensive
- Incredibly powerful
- **Finds bugs**

We don't expect you to use it, but getting used to automated testing makes you more useful in a team.

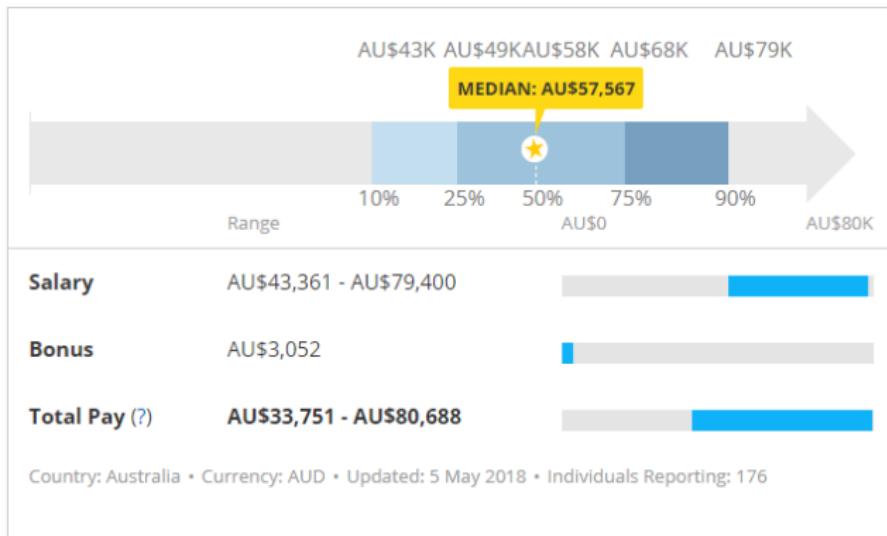
Here's [an example](#).

Assess Yourself

What units, use cases, and unit tests *could* you write for Project 2B?

Again, we don't expect you to do this.

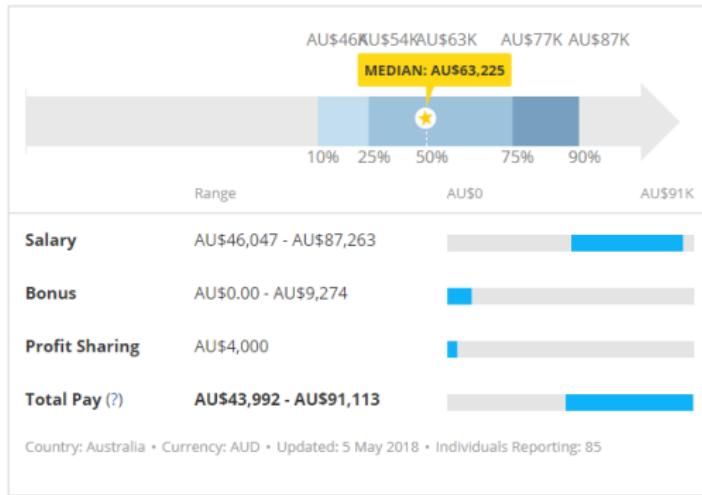
Software Testing and QA Jobs



Keyword

Software Tester: Conducts tests on software, primarily to find and eliminate bugs.

Software Testing and QA Jobs



Keyword

Software Quality Assurance: Actively works to improve the development process/lifecycle. Directs software testers to conduct tests, primarily to prevent bugs.

Metrics

Documentation

This will be assessed in the project, but not the exam.

Software Design

You will need to be able to define the *keywords* defined in this lecture. You will need to know *definitions* for the exam, but you will not be assessed on software design principles by writing code.

Software Testing

You will need to be able to define the keywords as well as implement a *unit test* for the exam. You will **not** be asked to write a TestRunner class, only one or two standalone test cases.

Assess Yourself

Describe the Observer pattern and the general “class” of problems it can be applied to.

Provide at least one real-world example where the Observer pattern could be applied.

Assess Yourself

The Observer pattern (also known as the *publish/subscribe* model) is used in situations where one or more classes (the observers) must *observe* and *react* to another object's (the subject) state.

Example: A robotic system has a number of sensors that are *observed* by the robot's central control system. When a sensor is *activated*, the central system is *notified* so the appropriate response can be taken.

SWEN20003
Object Oriented Software Development

Games and Event Driven Programming

Semester 1, 2020

The Road So Far

- Java Foundations
- Classes and Objects
- Abstraction
- Advanced Java
 - ▶ Exception Handling
 - ▶ Generic Classes
 - ▶ Generic Programming
 - ▶ Design Patterns
 - ▶ Software Testing and Design
- Software Development Tools

Lecture Objectives

After this lecture you will be able to:

- Describe the event-driven programming paradigm
- Describe where it can be applied, and how
- Implement basic event-driven programs in Java
- Describe some basic techniques in game design

Event Programming

How do programs work?

```
public static void main(String args[]) {  
  
    Scanner scanner = new Scanner(System.in);  
  
    String firstName = scanner.nextLine();  
    String lastName = scanner.nextLine();  
  
    String fullName = String.format("%s %s",
                                    firstName, lastName);  
  
    System.out.format("Welcome %s!\n", fullName);  
}
```

How do programs work?

Step 1: Create variable scanner

Step 2: Instantiate Scanner object and allocate to scanner variable

Step 3: Create variable firstName

Step 4: Call readLine method in scanner

Step 5: Allocate result to firstName

Step 6: ...

Step 7: ...

How do programs work?

Keyword

Sequential Programming: A program that is run (more or less) top to bottom, starting at the beginning of the `main` method, and concluding at its end.

- Useful for “static” programs
- Constant, unchangeable logic
- Execution is the same (or very similar) each time

Let's make our programs more “dynamic” ...

Event-Driven Programming

Keyword

State: The properties that define an object or device; for example, whether it is “active”.

Keyword

Event: Created when the *state* of an object/device/etc. is altered.

Keyword

Callback: A method triggered by an event.

Event-Driven Programming

Keyword

Event-Driven Programming: Using *events* and *callbacks* to control the flow of a program's execution.

Have we seen similar behaviour before?

- Exception handling
- Observer pattern

Event-Driven Programming

What makes this approach better?

Using an event-driven approach allows us to:

- Better *encapsulate* classes by hiding their behaviour
- Avoid having to explicitly send information about the input; instead it is automatically passed as part of the *callback*
- Easily add/remove behaviour to classes
- Easily add/remove additional *responses*

Assess Yourself

When playing a game, what *events* might we respond to?

- Mouse
- Keyboard
- Touch
- Controller (e.g. XBox controller)
- Time

Assuming we could respond to those events, what *features* could we add? Think about games you may have played, and how they work.

- Mouse: Menu buttons, GUI controls, click-to-move
- Keyboard: movement, “shooting”, special powers
- Touch/Controller: similar to keyboard
- Time: time-based completion, do-x-every-second (recover health, generate enemy)

The Event Loop

```
public class Robot {  
    public static void main(String args[]) {  
        while (true) {  
            if (isCommandSent()) {  
                respondToCommand();  
            } else if (isLowPower()) {  
                respondToLowPower();  
            } else if (isStuck()) {  
                respondToIsStuck();  
            } else if (hasCollided()) {  
                respondToCollision();  
            } else if (isSensorBroken()) {  
                respondToBrokenSensor();  
            } else if (...) {  
                ...  
            }  
        }  
    }  
}
```

What would you say is the “problem” in this code?

The Event Loop

Keyword

Polling: (also called *sampling*) relies on the program actively enquiring about the state of an object/device/etc.

What are some disadvantages of this approach?

- Lots of “waiting” for something to happen
- Lots of time/effort wasted “asking”
- Always responds in the same order
- Can’t “escape” from one method to respond to something else (potentially) urgent

Asynchronous Programming

Keyword

Interrupt: A signal generated by hardware or software indicating an event that needs *immediate* CPU attention.

Keyword

Interrupt Service Routine: Event-handling code to respond to interrupt signals.

Interrupts are used to control program flow at a *very low level*, **immediately** taking control of the program, e.g.:

- Exception/error handling
- Activating a sleeping process/task
- Device drivers (mouse, keyboard)

Robotics Case Study (Bad)

```
public class Robot {  
    public static void main(String args[]) {  
        while (true) {  
            if (isCommandSent()) {  
                respondToCommand();  
            } else if (isLowPower()) {  
                respondToLowPower();  
            } else if (isStuck()) {  
                respondToIsStuck();  
            } else if (hasCollided()) {  
                respondToCollision();  
            } else if (isSensorBroken()) {  
                respondToBrokenSensor();  
            } else if (...) {  
                ...  
            }  
        }  
    }  
}
```

Robotics Case Study (Good)

```
// Operates in a separate thread
public class DistanceSensor {

    public void detectCollision() {
        while (true) {
            // If a collision is imminent,
            // we should alert the main thread
            if (destructionImminent()) {
                mainThread.interrupt();
            }
        }
    }
}
```

Note: No one builds robots in Java.

Event-Driven Programming

Real-world examples:

- Graphical User Interfaces (GUIs)
- Web development/Javascript
- Embedded Systems/Hardware

Event-driven and *asynchronous* programming are very powerful techniques, and good tools to have in your arsenal.

Game Design

Inheritance-based Game Design

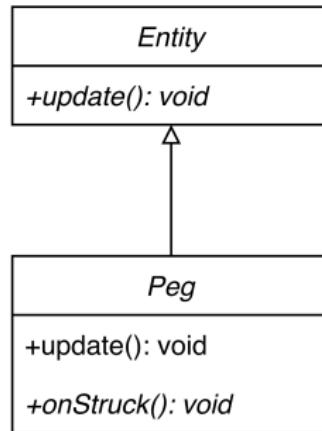
The most obvious way to design a game in an object-oriented language is to use inheritance. We could

- create an **Entity** abstract base class to represent game objects
- inherit from **Entity** to create new types of objects
- take advantage of **polymorphism**
- use the **Factory** design pattern

Inheritance

Many distinct Entitys may have similar behaviours.

All Pegs in *Shadow Bounce* have a behaviour when struck by a ball, so one obvious approach is a design like this:

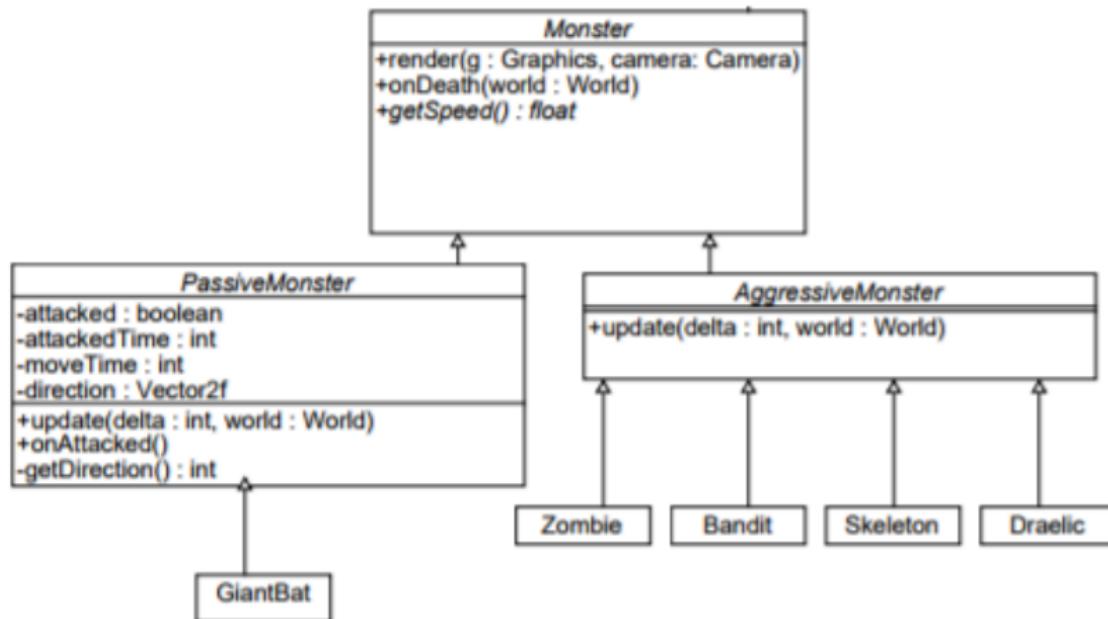


Zombies and bandits and bats, oh my!

You are designing a role-playing game, and you are told monsters come in two varieties; *aggressive* monsters that chase and attack the player, and *passive* monsters that simply try to run away.

You decide to implement this using inheritance.

UML diagram

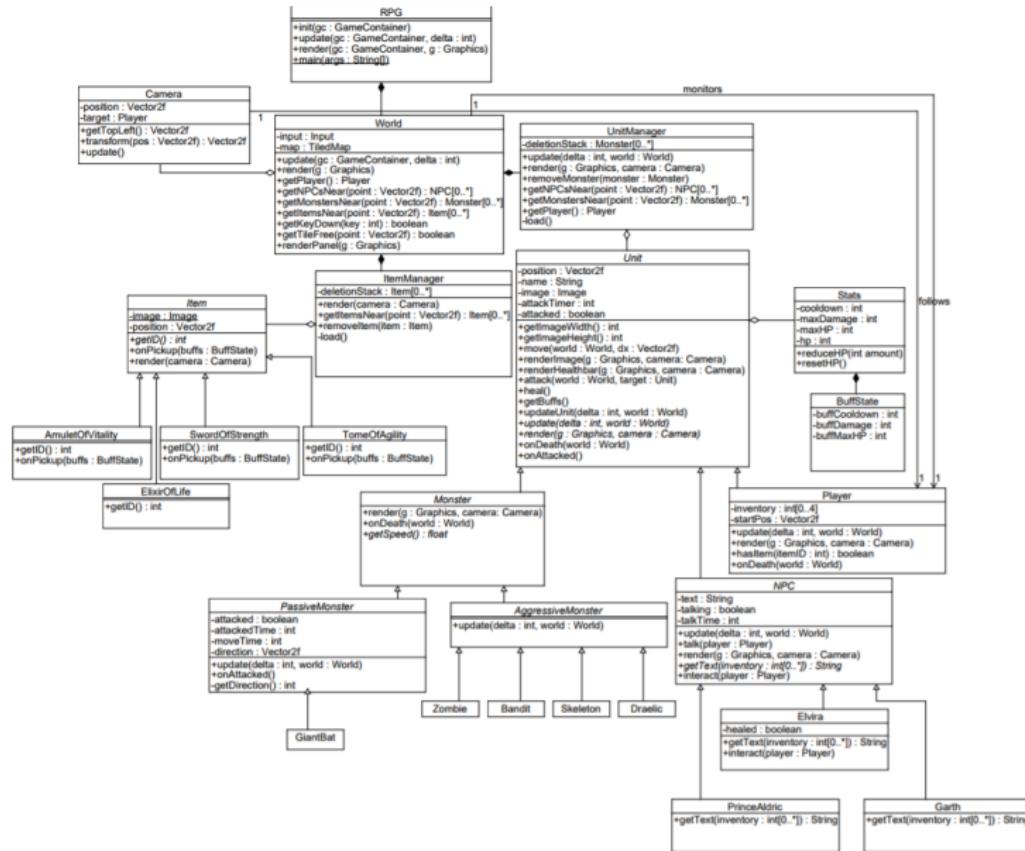


Pitfall: Flexibility

What if we wanted a monster that was sometimes passive and sometimes aggressive?

- inherit from both – can't do that...
- use an interface? Doesn't really fit...

An example game design



Pitfall: Complexity

This is a fairly small game, and that diagram is already looking pretty terrifying.

It's hard to tell from the small image, but there's also lots of data awkwardly passed around.

Solution: *Composition over inheritance*

Let's try a different approach. What if we broke down functionality of each object into its key Components? That is, those parts that describe it completely?

An Entity then becomes a **composition** of its Components.

A monster as a group of components

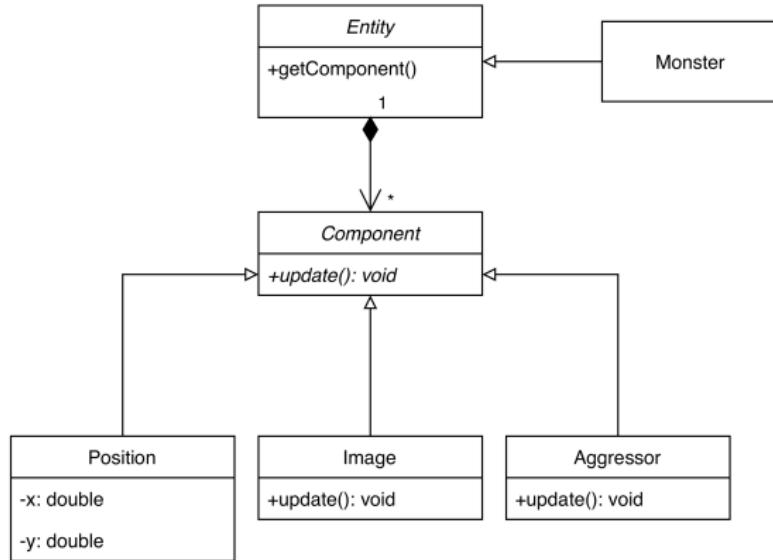
```
class Monster {  
    private Position position;  
    private Image image;  
    private Aggressor aggressor;  
  
    public void update() {  
        image.update(position);  
        aggressor.update(position);  
    }  
}
```

Fixing flexibility

Let's return to the problem of a Monster that may sometimes be aggressive, and sometimes not aggressive.

At the moment, the Monster is hard-coded to update its Aggressor component. We would like to be able to switch this on and off.

Monster's Components



Putting the pieces together

Let's write the classes. We need an Entity class, which is just a list of components.

```
class Entity {  
    private List<Component> components = new ArrayList<>();  
  
    public void addComponent(Component component) {  
        components.add(component);  
    }  
  
    public void update() {  
        for (Component c : components) {  
            if (c.getEnabled()) c.update();  
        }  
    }  
}
```

Continued...

Now we need an abstract Component that can be updated and rendered, as well as switched on and off.

```
abstract class Component {  
    private boolean enabled = true;  
  
    public boolean getEnabled() {  
        return enabled;  
    }  
  
    public void setEnabled(boolean enabled) {  
        this.enabled = enabled;  
    }  
  
    public abstract void update();  
}
```

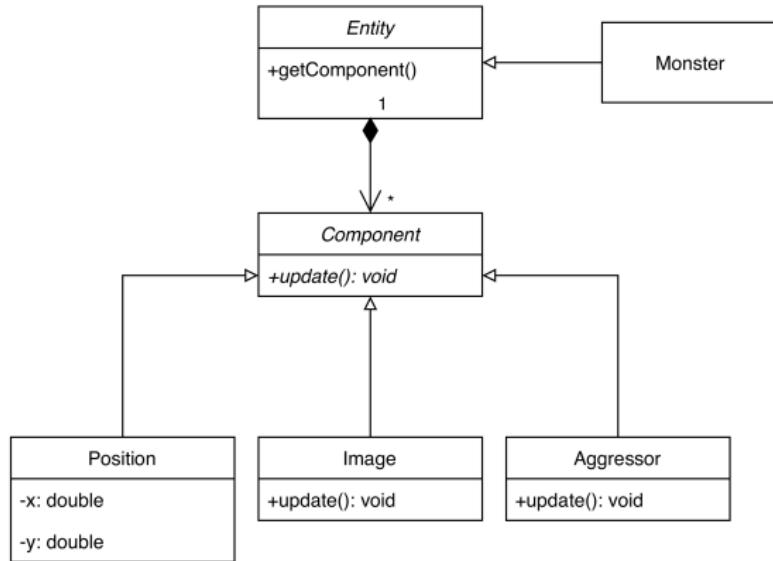
The road so far

```
Entity monster = new Entity();
monster.addComponent(new Position());
monster.addComponent(new Image());
monster.addComponent(new Aggressor());
```

By doing this, we have separated our complex inheritance diagram into Entity objects which are compositions of Component objects. As a bonus, we can easily reuse Components between different Entitys!

This is called **composition over inheritance**, and is an important tool to have in your arsenal.

Monster's Components



Components interacting with others

To implement an Image component, we need to look up the Position component. We would like to have something like this:

```
class Image extends Component {  
    private ImageRender image;  
  
    @Override  
    public void update() {  
        Position position = something.getComponent(Position);  
        image.draw(position.getX(), position.getY());  
    }  
}
```

What is this something? The most sensible choice is the Entity the component is attached to. (Why?)

Implementing lookup

We need to add some code to our Component base class:

```
abstract class Component {  
    private Entity entity;  
    public Entity getEntity() { return entity; }  
    public Component(Entity entity) { this.entity = entity; }  
}
```

Applying this in practice looks something like:

```
Entity monster = new Entity();  
monster.addComponent(new Position(monster));  
monster.addComponent(new Image(monster));  
monster.addComponent(new Aggressor(monster));
```

Components interacting with others

To implement an Image component, we need to look up the Position component. We would like to have something like this:

```
class Image extends Component {  
    private ImageRender image;  
  
    @Override  
    public void update() {  
        Position position = getEntity().getComponent(Position);  
        image.draw(position.getX(), position.getY());  
    }  
}
```

Now, how do we implement `getComponent`? We need to somehow pass a type as an argument.

Reflection

You might think to try using generics to solve this problem. Unfortunately, Java generics don't let us use e.g. `instanceof`.
However, Java lets us look up the class of an object!

```
class Entity {  
    public <T> T getComponent(Class<T> classObj) {  
        for (Component c : components) {  
            if (c.getClass().equals(classObj)) {  
                return (T) c;  
            }  
        }  
        return null;  
    }  
}
```

Implementing Image

This lets us do something like this:

```
class Image extends Component {  
    private ImageRender image;  
  
    @Override  
    public void update() {  
        Position position = getEntity()  
            .getComponent(Position.class);  
        image.draw(position.getX(), position.getY());  
    }  
}
```

Note: reflection is not examinable.

Summary

The **entity-component** approach is an example of a principle called **composition over inheritance**. This principle is a response to the problems caused by large, complex inheritance hierarchies, and tries to simplify things by using composition instead where possible.

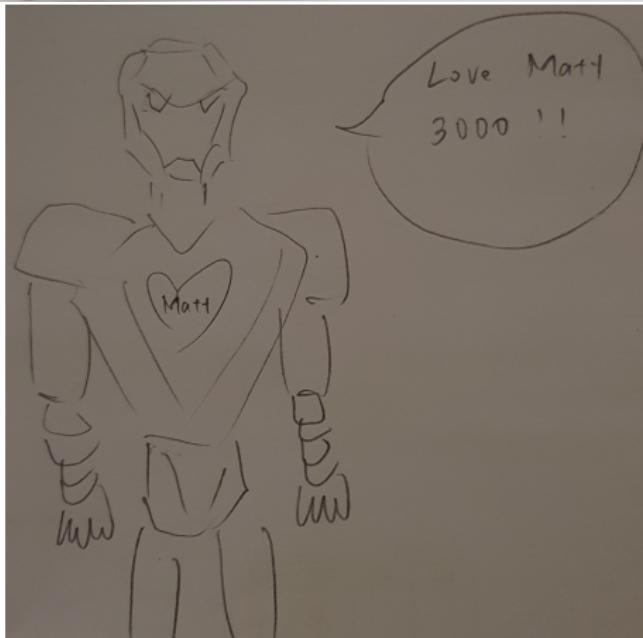
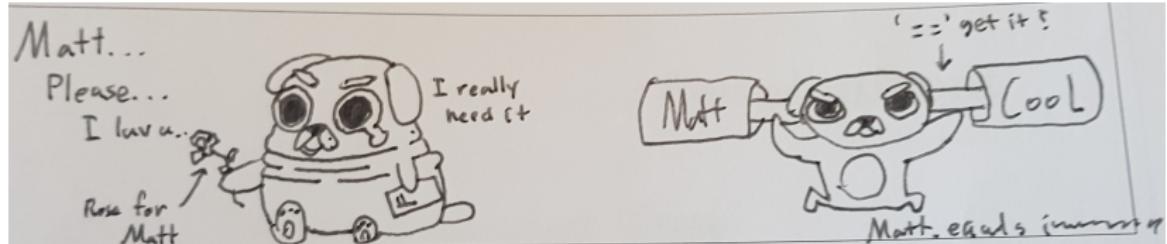
Questions?

Feel free to ask us about embedded programming or composition over inheritance.

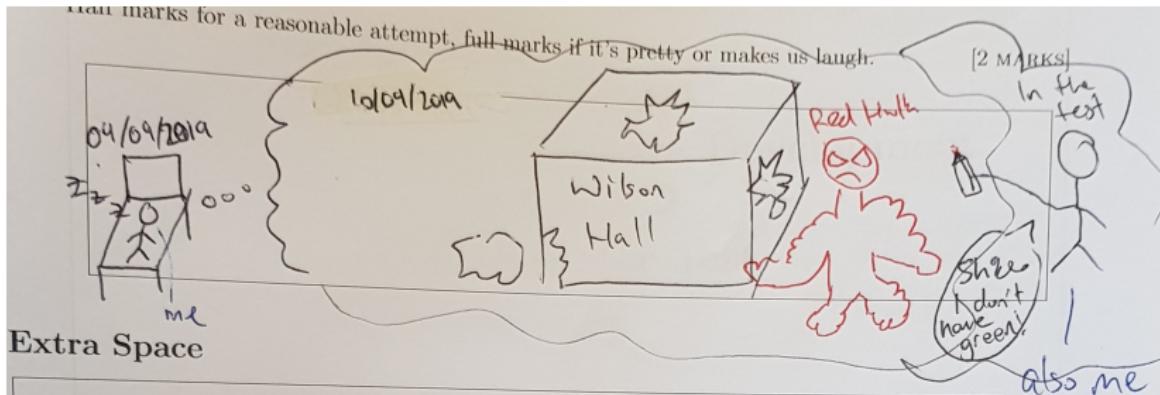
MST Drawings

The highlight reel can be found on [Google Drive!](#)

MST Drawings - The Cute



MST Drawings - The Good



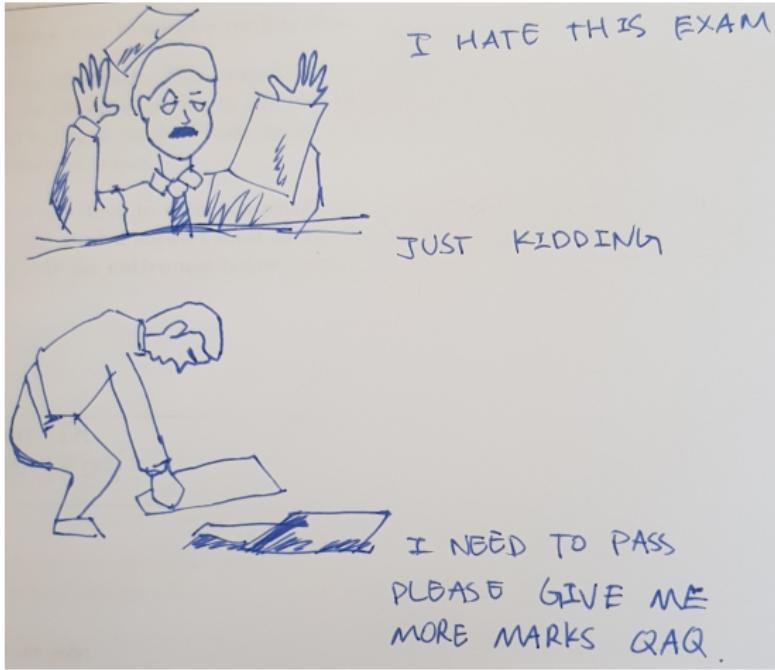
Context of the drawing!

I'm a bad ~~draw~~-artist so here's an explanation.

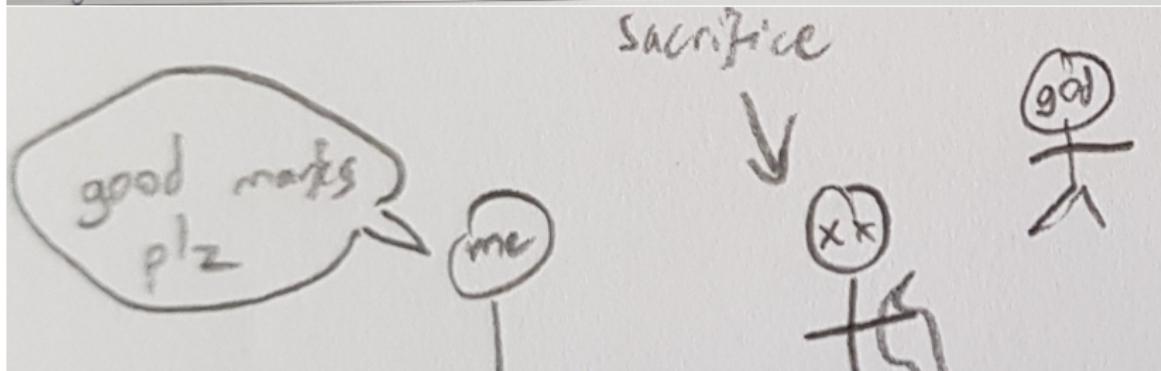
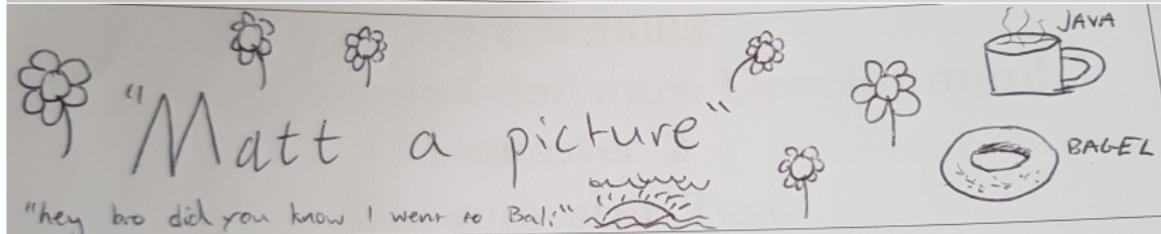
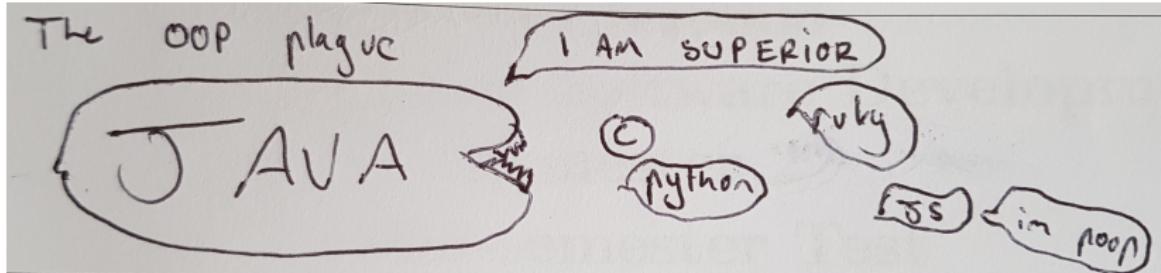
This is basically me dreaming the night before the test about me drawing the drawing to Q4.

The drawing that I'm drawing in the dream is a drawing of me drawing red hulk (red cause I didn't have green) destroying Wilson Hall.

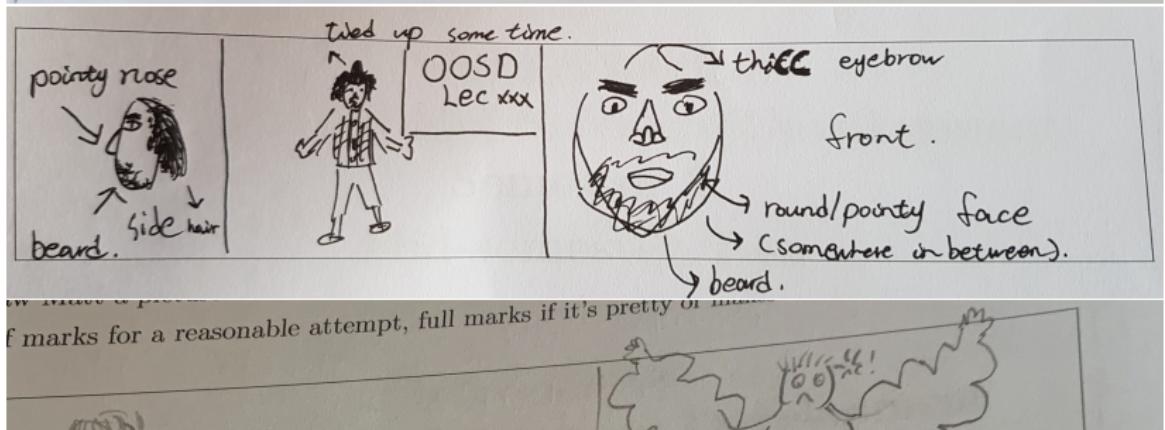
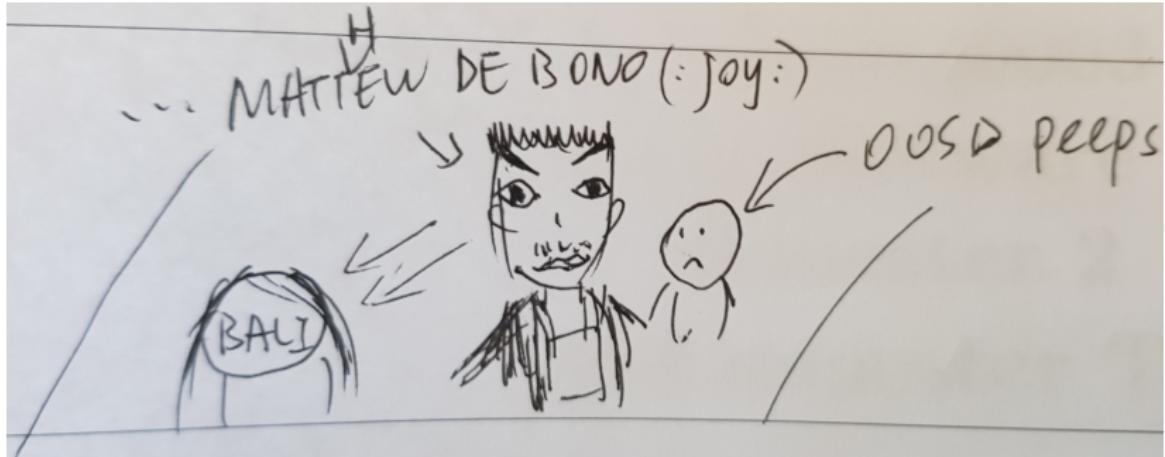
MST Drawings - The Sad



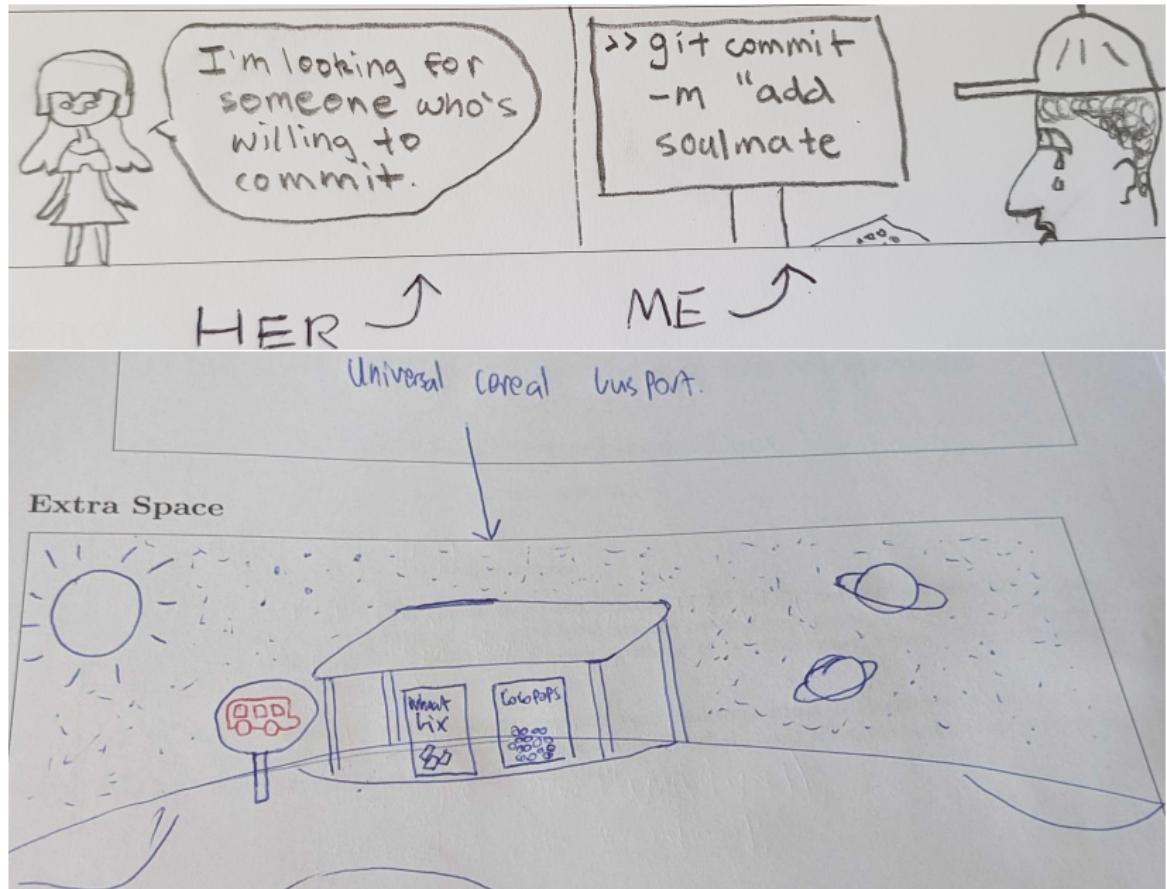
MST Drawings - The Random



MST Drawings - The Tributes to Matt



MST Drawings - The Funny



Metrics

You should be able to explain the various keywords and concepts introduced in this lecture, including:

- Sequential, event-driven, and asynchronous programming
- Examples of event-driven systems
- Basic game design, such as the Entity Component approach

Given the appropriate tools/documentation in the exam, you should also be able to:

- Analyse an event-driven system, to determine its behaviour
- Implement event-handling components in an event-driven system

Assess Yourself

Sample worded exam questions:

- ① Describe how sequential programming differs from asynchronous programming.
- ② Describe the event-driven programming paradigm.
- ③ How does the observer pattern demonstrate event-driven programming?
- ④ Explain (with examples) some of the downsides of using object-oriented programming for game development.
- ⑤ Describe the Entity-Component approach to game development.

SWEN20003
Object Oriented Software Development

Advanced Java and OOP

Semester 1, 2020

The Road So Far

- Java Foundations
- Classes and Objects
- Abstraction
- Advanced Java
 - ▶ Exception Handling
 - ▶ Generic Classes
 - ▶ Generic Programming
 - ▶ Design Patterns
 - ▶ Software Testing and Design
 - ▶ Games and Events
- Software Development Tools

Lecture Objectives

After this lecture you will be able to:

- Describe and use enumerated types
- Make use of functional interfaces and lambda expressions
- Do cool stuff in Java

Enumerated Types

Assess Yourself

You've been hired by gambling company *Soulless* to design and implement their newest card game called *Horses*.

Soulless have asked you to build a preliminary design before telling you the rules of *Horses*.

How would you design this game, knowing only that you are implementing a card game.

Assess Yourself

Problem: How do you represent a Card class?

A Card consists of a Suit, Rank, and Colour.

Okay... How do we represent those?

Enumerated Types

Keyword

`enum`: A class that consists of a **finite** list of constants.

- Used any time we need to represent a fixed set of values
- Must list *all* values
- Otherwise, just like any other class; they can have methods and attributes!

Let's define the Card class and the Rank enum.

Defining a Card

```
public class Card {  
  
    private Rank rank;  
    private Suit suit;  
    private Colour colour;  
  
    public Card(Rank rank, Suit suit, Colour colour) {  
        this.rank = rank;  
        this.suit = suit;  
        this.colour = colour;  
    }  
}
```

Defining a Card

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

What does it do? How would you expect to **use** it?

Enum Variables

```
Rank rank = Rank.ACE;  
Card card = new Card(Rank.FOUR, ..., ...);
```

The values of an enum are accessed *statically*, because they are constants.

Enum objects are treated just like any other object.

Let's make the other components...

Defining a Card

```
public enum Colour {  
    RED, BLACK  
}
```

```
public enum Suit {  
    SPADES, CLUBS, DIAMONDS, HEARTS  
}
```

Can anyone see a flaw in our Card design? Any assumptions we've made/not made?

Shouldn't the Colour and Suit be related in some way?

Defining a Card

```
public enum Suit {  
    SPADES(Colour.BLACK),  
    CLUBS(Colour.BLACK),  
    DIAMONDS(Colour.RED),  
    HEARTS(Colour.RED);  
  
    private Colour colour;  
  
    private Suit(Colour colour) {  
        this.colour = colour;  
    }  
}
```

Now, every Suit is automatically tied to the appropriate Colour; this *may or may not* be useful behaviour.

Enum Variables

```
public static void main(String args[]) {  
    ArrayList<Rank> ranks = new ArrayList<>();  
  
    ranks.add(Rank.TEN);  
    ranks.add(Rank.FOUR);  
    ranks.add(Rank.EIGHT);  
    ranks.add(Rank.THREE);  
    ranks.add(Rank.ACE);  
  
    System.out.println(ranks);  
    Collections.sort(ranks);  
    System.out.println(ranks);  
}
```

[TEN, FOUR, EIGHT, THREE, ACE]
[ACE, THREE, FOUR, EIGHT, TEN]

Enum Variables

Enums come pre-built with...

- Default constructor
- `toString()`
- `compareTo()`
- `ordinal()`

Enums are also *classes*, so we can add (or override) any method or attribute we like.

```
public boolean isFaceCard() {  
    return this.ordinal() > Rank.TEN.ordinal();  
}
```

Assess Yourself

What is an enum?

What other applications can you think of for them?

Variadic Parameters

What?

```
List<Integer> list = Arrays.asList(12, 5);
```

```
List<Integer> list = Arrays.asList(12, 5, 45, 18);
```

```
List<Integer> list = Arrays.asList(12, 5, 45, 18, 33);
```

How does this method work? Is it overloaded for any number of arguments...?

There is a better way!.

Variadic Parameters

Keyword

Variadic Method: A method that takes an *unknown number* of arguments.

```
public String concatenate(String... strings) {  
    String string = "";  
  
    for (String s : strings) {  
        string += s;  
    }  
  
    return string;  
}
```

Variadic methods *implicitly* convert the input arguments into an array. Be careful!

Assess Yourself

Write a variadic method that computes the average of an unknown number of integers.

```
public double average(int... nums) {  
    int total = 0;  
  
    for (int i : nums) {  
        total += i;  
    }  
  
    return 1.0 * total / nums.length;  
}
```

```
System.out.println(average(1));  
System.out.println(average(1, 2, 3));  
System.out.println(average(10, 20, 30, 40, 50));
```

1.0
2.0
30.0

Functional Interfaces

Functional Interfaces

Keyword

Functional Interface: An interface that contains only a single abstract method; also called a Single Abstract Method interface.

```
@FunctionalInterface  
public interface Attackable {  
    public void attack();  
}
```

Functional interfaces can contain only one “new” non-static method; adding more will raise an error.

Functional Interfaces

Cool story... But... Why?

Functional interfaces are a *tool* that we can use with other techniques...

But let's look at a few functional interfaces first.

Functional Interfaces

```
public interface Predicate<T>
```

The Predicate functional interface...

- Represents a *predicate*, a function that accepts one argument, and returns `true` or `false`
- Executes the `boolean test(T t)` method on a single object
- Can be combined with other predicates using the `and`, `or`, and `negate` methods

Functional Interfaces

```
public interface UnaryOperator<T>
```

The `UnaryOperator` functional interface...

- Represents a *unary* (single argument) function that accepts one argument, and returns an object of the same type
- Executes the `T apply(T t)` method on a single object

Lambda Expressions

Lambda Expressions

Keyword

Lambda Expression: A technique that treats code as *data* that can be used as an “object”; for example, allows us to *instantiate* an interface without implementing it.

```
public interface Predicate<T>
```

```
Predicate<Integer> p = i -> i > 0;
```

The Predicate functional interface is now an *object* that implements the function to test if integers are greater than zero.

Lambda Expressions

```
(sourceVariable1, sourceVariable2, ...)  
    -> <operation on source variables>
```

A lambda expression takes zero or more arguments (source variables) and applies an operation to them

Operations could be:

- Doubling an integer
- Comparing two objects
- Performing a boolean test on an object
- Copying an object
- ...

Assess Yourself

What does this code do?

```
Predicate<Integer> p1 = i -> i > 0;
Predicate<Integer> p2 = i -> i%2 == 0;
Predicate<Integer> p3 = p1.and(p2);

List<Integer> nums = Arrays.asList(1, 2, 5, 6, 7, 4, 5);

for (Integer i : nums) {
    if (p3.test(i)) {
        System.out.println(i);
    }
}
```

2
6
4

Assess Yourself

```
public abstract class List<T> {  
    public void replaceAll(UnaryOperator<T> operator);  
}
```

```
List<String> names = Arrays.asList("Tony", "Thor", "Thanos");  
  
names.replaceAll(name -> name.toUpperCase());  
  
System.out.println(names);
```

```
["TONY", "THOR", "THANOS"]
```

Anonymous Classes vs. Lambdas

Lambda expressions can often be used *in place* of anonymous classes, but are not the same thing.

Anonymous Class

```
starWarsMovies.sort(new Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {
        return m1.rating - m2.rating;
    }
});
```

Lambda Expression

```
starWarsMovies.sort((m1, m2) -> m1.rating - m2.rating);
```

Lambda Expressions

Lambda expressions are *instances of functional interfaces*, that allow us to treat the functionality of the interface as an *object*.

This makes our code **much** neater, and easier to read.

What next?

Method References

Rewind a Bit

```
List<String> names = Arrays.asList("Tony", "Thor", "Thanos");  
names.replaceAll(name -> name.toUpperCase());  
System.out.println(names);
```

What does this code do?

How would you describe the *effect* of the lambda expressions?

The lambda expression *applies one method* to every element of the list.
We can take this a step further...

Method References

```
names.replaceAll(String::toUpperCase);
```

Keyword

Method Reference: An object that stores a *method*; can take the place of a lambda expression **if** that lambda expression is only used to call a single method.

Method references can be *stored* in the same way a lambda expression can:

```
UnaryOperator<String> operator = s -> s.toLowerCase();
```

```
UnaryOperator<String> operator = String::toLowerCase;
```

Method Reference Examples

Static methods:

```
Class::staticMethod  
Person::printWarning
```

Instance methods:

```
Class::instanceMethod || object::instanceMethod  
String::startsWith || person::toString
```

Constructor:

```
Class::new  
String::new
```

Method arguments are now *implied*, and given when the method is *called*.

Method Reference Examples

```
public class Numbers {  
    public static boolean isOdd(int n) {  
        return n % 2 != 0;  
    }  
}
```

```
public static List<Integer> findNumbers(  
    List<Integer> list, Predicate<Integer> p) {  
    List<Integer> newList = new ArrayList<>();  
  
    for(Integer i : list) {  
        if(p.test(i)) {  
            newList.add(i);  
        }  
    }  
  
    return newList;  
}
```

Method Reference Examples

```
List<Integer> list = Arrays.asList(12, 5, 45, 18, 33, 24, 40);

// Using an anonymous class
findNumbers(list, new Predicate<Integer>() {
    public boolean test(Integer i) {
        return Numbers.isOdd(i);
    }
});

// Using a lambda expression
findNumbers(list, i -> Numbers.isOdd(i));

// Using a method reference
findNumbers(list, Numbers::isOdd);
```

Streams

Assess Yourself

Write a function that accepts a list of String objects, and returns a *new* list that contains only the Strings with at least five characters, starting with "C". The elements in the new list should all be in *upper case*.

```
public List<String> findElements(List<String> strings) {  
    List<String> newStrings = new ArrayList<>();  
  
    for (String s : strings) {  
        if (s.length() >= 5 && s.startsWith("C")) {  
            newStrings.add(s.toUpperCase());  
        }  
    }  
}
```

Motivation

Now that we have these fancy new tools, what can we do with them?

What if we wanted to apply *multiple* functions to the same data?

That's where streams come in!

Keyword

Stream: A series of elements given in *sequence*, that are *automatically* put through a *pipeline* of operations.

Using Streams

We can think of that example as applying a sequence of operations to our list:

- Iterating through the list...
- Selecting elements with length greater than five...
- And elements with first character “C” ...
- Then, converting those elements to upper case...
- And adding them to a new list

```
list = list.stream()
            .filter(s -> s.length() > 5)
            .filter(s -> s.startsWith("C"))
            .map(String::toUpperCase)
            .collect(Collectors.toList());
```

Streams

Streams are a powerful Java technique that allow you to apply *sequential* operations to a collection of data. These operations include:

- map (convert input to output)
- filter (select elements with a condition)
- limit (perform a maximum number of iterations)
- collect (gather all elements and output in a list, array, String...)
- reduce (*aggregate* a stream into a single *value*)

Given this...

Assess Yourself

Implement a stream pipeline that takes a list of Person objects, and generates a String consisting of a comma separated list.

The list should contain the names (in upper case) of all the people who are between the ages of 18 and 40.

```
List<Person> people = Arrays.asList(
        new Person("Peter Parker", 18),
        new Person("Black Widow", 34),
        new Person("Thor", 1500),
        new Person("Nick Fury", 67),
        new Person("Iron Man", 49)
);
```

Assess Yourself

Implement a stream pipeline that takes a list of People, and generates a String consisting of a comma separated list.

The list should contain the names (in upper case) of all the people who are between the ages of 18 and 40.

```
String output = people.stream()
    .filter(p -> p.getAge() >= 18)
    .filter(p -> p.getAge() <= 40)
    .map(Person::getName)
    .map(String::toUpperCase)
    .collect(Collectors.joining(", "));
```

"PETER PARKER, BLACK WIDOW"

Metrics

- You should be able to conceptually describe all of the techniques presented in this lecture.
- You should be able to *read* and *interpret* code using any of the techniques in this lecture.
- You will **not** be expected to **write** code on anything from today.

SWEN20003
Object Oriented Software Development

Final Exam and Beyond

Semester 1, 2020

Topics Covered During the Semester

- Subject Introduction
- A Quick Tour of Java
- Classes and Objects
- Arrays and Strings
- Software Tools
- Input and Output
- Inheritance and Polymorphism
- Interfaces and Polymorphism
- Modelling Class Relationships with UML
- Generics
- Exceptions
- Design Patterns
- Software Testing and Design
- Asynchronous Programming
- Advanced Java and OOP

Lecture Objectives

- Review the subject learning outcomes
- Plan your future COMP/SWEN subjects
- Discuss the exam structure and process
- Help you do well in the exam

A Quick Tour of Java

Learning Outcomes:

- Identify some of the key Java features
- Understand the following in context of Java:
 - ▶ Identifiers, Data Types, Variables and Constants
 - ▶ Operators and Expressions
 - ▶ Flow of control
- Write simple Java programs

Classes and Objects

Learning Outcomes:

- Explain the difference between a *class* and an **object**
- Define classes, and give them **properties** and **behaviours**
- Implement and use classes
- Identify a series of well-defined classes from a **specification**
- Explain object oriented concepts: **abstraction**, **encapsulation**, **information hiding** and **delegation**
- Understand the role of **Wrapper** classes

Arrays and Strings

Learning Outcomes:

- Understand how to use **Arrays**
- Understand how to use **Strings**

Software Tools

Learning Outcomes:

- How to use a **version control system** to manage your software
- How to use the software development framework **Bagel**

Input and Output

Learning Outcomes:

- Accept input to your programs through:
 - ▶ Command line arguments
 - ▶ User input
 - ▶ Files
- Write output from your programs through:
 - ▶ Standard output (terminal)
 - ▶ Files
- Use files to store and retrieve data during program execution
- Manipulate data in files (i.e. for computation)

Inheritance and Polymorphism

Learning Outcomes:

- Use **inheritance** to abstract common properties of classes
- Explain the relationship between a **superclass** and a **subclass**
- Make better use of **privacy** and **information hiding**
- Identify errors caused by **shadowing** and **privacy leaks**, and avoid them
- Describe and use method **overriding**
- Describe the **Object** class, and the properties inherited from it
- Describe what **upcasting** and **downcasting** are, and when they would be used
- Explain **polymorphism**, and how it is used in Java
- Describe the purpose and meaning of an **abstract** class

Interfaces and Polymorphism

Learning Outcomes:

- Describe the purpose and use of an **interface**
- Describe what it means for a class to **use** an interface
- Describe when it is appropriate to use inheritance vs. interfaces
- Use interfaces and inheritance to achieve powerful abstractions
- Make any class “sortable”

Modelling Class Relationships with UML

Learning Outcomes:

- Identify **classes** and **relationships** for a given problem specification
- Represent classes and relationships in **UML**
- Develop simple **Class Diagrams**

Generics

Learning Outcomes:

- Understand **generic** classes in Java
- Use **generically typed** classes
- Define **generically typed** objects
- Choose appropriate data structures **storing, retrieving** and **manipulating** objects (data)
- Use the Java **Collections** Framework
- Use the Java **Maps** Framework

Exceptions

Learning Outcomes:

- Understand what **exceptions** are
- Appropriately handle **exceptions** in Java
- Define and utilise **exceptions** in Java

Design Patterns

Learning Outcomes:

- Describe what **design patterns** are, and why they are useful
- Analyse and understand **design pattern specifications**
- Make use of (some) design patterns

Software Testing and Design

Learning Outcomes:

- **Write better** code
- **Design better** software
- **Test** your software for bugs

Asynchronous Programming

Learning Outcomes:

- Describe the **event-driven programming** paradigm
- Describe where it can be applied, and how
- Implement basic event-driven programs in Java
- Describe some basic techniques in game design

Learning Outcomes:

- Describe and use enumerated types
- Make use of functional interfaces and lambda expressions
- Do cool stuff in Java

Assessment

Mark distribution across assessment components:

- Workshops: 10%
- Project: 50%
- Final Exam: 40%

Note: There is a 50% hurdle on each component

What I Hope You Learned

- How to program in Java
- How to think like a “designer”
- Objected-oriented programming (Java or otherwise) is a powerful tool
- Software design is complicated, and involves a lot of tools, techniques, and tradeoffs
- Game development is fun
- On-line learning is not too bad
- We have the resilience and ability to adapt - covid-19 being the case study

What Did We Miss?

- More advanced Java (e.g. multithreading)
- There is plenty of UML beyond class diagrams:
 - ▶ Activity diagrams
 - ▶ Sequence diagrams
 - ▶ Object diagrams
 - ▶ And more...
- There are plenty more design patterns
- We barely scratched the surface of software design tools, and techniques

Future Subjects

- Software Modelling and Design (SWEN30006)
 - ▶ More UML, design patterns, software design techniques
 - ▶ More Java
- Computer Systems (COMP30023)
 - ▶ How does computer hardware work and interact; uses C
- IT Project (COMP30022)
 - ▶ Capstone project for the major
 - ▶ Java is *assumed*, not taught

The Exam

Exam Structure

- The exam has three sections:
 - ▶ Section 1 - Short Answer (20 marks)
 - ▶ Section 2 - System Design (30 marks)
 - ▶ Section 3 - Java Development (50 marks)
- 100 marks (weighted to 40% of the final mark)
- Allocated Time: 120 minutes

Note: This includes reading time if you want to use it for that purpose, but you can start answering the questions during this time if you choose to do so.

How to access and submit the exam

- Go to the subject on Canvas
- Select the "Assignments" tab
- Download your exam paper in .pdf format from: "Semester 1 exams, 2020"
- Read and carefully follow the "Instructions to students" section on the exam paper
- Submit the exam, via 'Semester 1 exams, 2020" link on Canvas

Instructions to students

- The exam has 4 questions across 3 sections, and all questions must be attempted.
- This is an online, open-book exam and you will have access to your subject material and Internet resources.
- You are required to provide your own answers to questions; copying directly from subject material or Internet sources is not acceptable and will be considered plagiarism.
- The mark allocation for each question is listed along with the question. Please use the marks as a guide to the detail required in your answers while keeping your answers concise and relevant. Point form is acceptable in answering descriptive questions. Any unreadable answers will be considered wrong.

Instructions to students contd..

- Worded questions must all be answered in English, and code questions must all be answered in Java.
- Please follow the instructions below to complete the exam:
 - ▶ Clone the solution template from the repository <username>-final-exam in gitlab.eng.unimelb.edu.au using the appropriate: git clone command.
 - ▶ In this repository you will find a single file named <username>-final-exam.txt. You are required to provide the answers to questions in the appropriate section of this file, through typing them in; all answers must be in this file and no other files should be added to the repository. You are not permitted to upload hand-written scanned files, and such solutions will receive zero marks.

Instructions to students contd..

- contd...
 - ▶ Upon attempting each sub-question commit the changes to your files (answers) using:
git add . and
git commit -m '[commit message]' performed from the cloned folder in the above step.
e.g. Once you complete Question 1(a) you can execute the following commands to commit the changes:
git add .
git commit -m 'completed question 1(a)'

- ▶ You may choose to do git push as often as you would like as a way of backing up, but are not required to push until you complete the exam.
- ▶ At the end of the exam, you are also required to submit the solution file, <username>-final-exam.txt, as a Canvas Turnitin Assignment submission.

Note: It is important to carefully follow the instructions above. Abnormal commits will receive scrutiny because it could indicate possible plagiarism.

Instructions to students contd..

- Your lecturers will be available online via Zoom, during the first 15 minutes of the exam (Zoom link will be made available via Canvas). During this time you may ask clarification questions. You will be allowed to start answering questions during this time if you choose to do so (this is not a strict reading time as in normal written exams)

Final Exam Tips

Finally...

- Read instructions carefully
- Try to read **all** the questions before writing anything.
- Exam is just testing what you have been doing throughout the semester
 - ▶ If you have been engaged with the subject through following the lectures, doing the workshops and assignments you should not have problems in the exam
 - ▶ However, you will need to have your thinking hats on because questions are about applying your knowledge

Final Exam Tips

Attempt the practice exam:

- Follow the instructions provided in “Instructions to students” which will be very similar to what you have to do before the exam.
- I will not provide a sample solution to the Practice Exam - this is for you to do by yourself.
- If there are any issues with submitting to git etc. please post questions on the discussion forum.

Important

- Please pay attention to subject announcements sent via Canvas after this week.
 - ▶ If there are any changes to what I have said so far, these will be announced Canvas announcements.
- Before the exam (at least a few days), I will make an announcement asking you to:
 - ▶ checkout the solution template, <username>-final-exam.txt from the repository;
 - ▶ add your student ID to the solution template and commit the file; and
 - ▶ push the file back to the repository.

If it does not work let us know via the discussion board and we can help you.

- Why is following the above steps important?
 - ▶ This will allow you to try the process before the exam and verify if your repositories are working as expected so that you will not have to stress out about this during the exam.

Questions?

SWEN20003
Object Oriented Software Development

Classes and Objects

Semester 1, 2020

The Road So Far

Lectures

- Subject Introduction
- A Quick Tour of Java

Learning Outcomes

Upon completion of this topic you will be able to:

- Explain the difference between a *class* and an *object*
- Define classes, and give them *properties* and *behaviours*
- Implement and use classes
- Identify a series of well-defined classes from a *specification*
- Explain object oriented concepts: abstraction, encapsulation, information hiding and delegation
- Understand the role of Wrapper classes

Introduction

All programming languages support four basic concepts:

- Calculation: constants, variables, operators, expressions
- Selection: `if-else`, `switch`, ?
- Iteration: `while`, `do`, `for`
- **Abstraction:** The process of creating self-contained units of software that allows the solution to be parameterized and therefore more general purpose

Abstraction is the fundamental concept that differentiates procedural programming languages such as C from Object Oriented languages such as Java, C++.

Abstraction in Procedural Languages

Abstraction in procedural languages is provided through *functions* or *procedures*.

Functions manipulate external data to by performing operations on them.

Example of a function in C that calculates the average of two floating point numbers:

```
float calculate_average (float a, float b) {  
    float result;  
    result = (a + b)/2;  
    return result;  
}
```

Abstraction in Object Oriented Languages

Abstraction in Object Oriented (OO) languages is provided through an *Abstract Data Type (ADT)*, which contains *data* and *functions* that operate on data.

In Java a **Class** is an implementation on an **Abstract Data Type**.

Classes

- A “generalization” of a real world (or “problem world”) entity
 - ▶ A physical real world thing, like a student or book
 - ▶ An abstract real world thing, like a university subject
 - ▶ An even more abstract thing like a list or a string (data)
- Represents a template for things that have common properties
- Contains *attributes* and *methods*
- Defines a new **data type**

Keyword

Class: Fundamental unit of abstraction in *Object Oriented Programming*.
Represents an “entity” that is part of a problem.

Objects

- Are an *instance* of a class
- Contain **state**, or dynamic information
- “**X** is of type A”, “**X** is an object of the class A”, and “**X** is an instance of the class A” are all equivalent

Keyword

Object: A specific, concrete example of a class

Keyword

Instance: An object that exists in your code

Motivating Example

Throughout this topic we will be referring to the following specification:

Develop a system (a set of classes) for a simple Drawing Pad application. The application should allow drawing different types of shapes, such as circles, squares, rectangles, display their geometrical properties: e.g. area, circumference. It should also allow different types of actions such as moving, resizing to the performed on shapes.

How would you develop this, right now? What additional information do you need?

Drawing Pad Application - Classes

What classes can we use for our example problem?

Fundamental:

- Drawing Pad
- Circle
- Square
- Other shapes

Additional:

- Drawing Tool
- Paint Brush
- Fill Colour
- Fill Type
- Many more

Identifying Attributes and Methods

Let us consider the `Circle` class.

Can you identify the *attributes* and *methods* of the class?

Attributes:

- Centre
- Radius
- Fill Colour, Fill Type
- Many more

Methods (Operations):

- Compute Circumference
- Compute Area
- Move
- Resize
- Many more

Object Oriented Features

Following are some key features of the object oriented design paradigm:

- Data Abstraction
- Encapsulation
- Information Hiding
- Delegation
- Inheritance
- Polymorphism

Data Abstraction

Keyword

Data Abstraction: The technique of creating new data types that are well suited to an application by defining new classes.

A class is a special kind of programmer-defined data type.

- For example, by creating classes such as Circle, Drawing Pad, you are creating new data types, that can be used in applications.

A class is somewhat close to a structure in C but has additional features - attributes and methods.

The class definition determines the types of data (attributes) that an object can contain, as well as the actions (methods) it can perform.

Encapsulation

Keyword

Encapsulation: The ability to group data (attributes) and methods that manipulate the data to a single entity though defining a class.

A class encapsulates data and the methods that operate on the data into a single unit.

This method of encapsulation is unique to OO programming and is not provided by the procedural programming paradigm.

Defining a Class

Defining a Class

Syntax:

```
<visibility modifier> class <ClassName> {  
    <attribute declarations>  
    <method declarations>  
}
```

A bare bone class:

```
// Circle.java - Circle class definition  
public class Circle {  
}
```

Defining a Class - Adding Attributes

Attribute Syntax:

```
<visibility modifier> <type> <variable name>;
```

Adding attributes (also called data, fields) to the Circle class.

```
// Circle.java - Circle class definition
public class Circle {
    public double centreX; //centre x coordinate
    public double centreY; //centre y coordinate
    public double radius; //radius
}
```

Defining a Class - Adding Attributes

The attributes added to the Circle class in the above example are referred to as *instance variables*.

- these attributes maintain the **state** of the object; i.e. by giving values to centreX, centreY, radius we define a Circle object with particular size and position.

Keyword

Instance Variable: A **property** or **attribute** that is unique to each *instance* (*object*) of a class, which maintains the **state** of the object.

Defining a Class - Adding Methods

Method Syntax:

```
<visibility modifier> <void or typeReturned> myMethod(paramList)
{
    variable declarations
    statements
}
```

- If the method returns data, the data type must be specified in the method definition, otherwise, it is defined as **void**.
- If the method returns data, the method body must contain a return statement, which returns a variable of the specified return type.
- Variables can be declared inside the method - such variables are called *local variables*.

Note: Local variables are inside the method as opposed to the instance variables (introduced earlier) which are outside the method declaration.

Defining a Class - Adding Methods

Adding methods to Circle class.

```
// Circle.java
public class Circle {
    public double centreX;
    public double centreY;
    public double radius;

    public double computeCircumference () {
        double circum = 2 * Math.PI * radius;
        return circum;
    }
    public double computeArea () {
        double area = Math.PI * radius * radius;
        return area;
    }
    public void resize (double factor) {
        radius = radius * factor;
    }
}
```

Using a Class

Using the Circle class

Follow the steps below to use the `Circle` class we just created.

- Create a file `Circle.java` and write the code.
Note: the file name should match the class name.
- You can use an Integrated Development Environment (IDE), such as IntelliJ for this (will be introduced in the workshops), but in this instance you can use a text editor such as notepad, wordpad, vim, kate etc.
- Compile the class using the following command:

```
javac Circle.java
```

This creates a file `Circle.class`

- `Circle` becomes a derived data type that can be used in a Java program.

Using the Circle class

By creating the Circle class, you have created a new data type **Circle - Data Abstraction**.

- Variables of type Circle can be now defined in a program
- Circle is a **Derived Data Type** (as opposed to a Primitive Data Type such as int, float)

Example:

```
/* CircleTest.java: A test program to test the Circle class */
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle;
        Circle bCircle
    }
}
```

Using the Circle class

The declarations:

```
Circle aCircle;  
Circle bCircle;
```

in the previous example did not create Circle objects.

aCircle and bCircle are simply **references** to Circle objects (not objects):

- Currently they point to nothing, hence **null references**.

aCircle



bCircle



Points to nothing
(Null Reference)

Points to nothing
(Null Reference)

The `null` Reference

Keyword

null: The Java keyword for “no object here”. Null objects can’t be “accessed” to get variables or methods, or used in any way.

Instantiating a Class

Objects are **null** until they are *instantiated*.

Keyword

Instantiate: To create an object of a class

```
// Instantiate an Circle object  
Circle circle_1 = new Circle();
```

Keyword

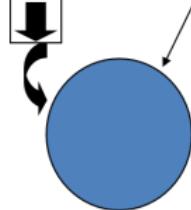
new: Directs the JVM to allocate memory for an object, or *instantiate* it

Creating Objects

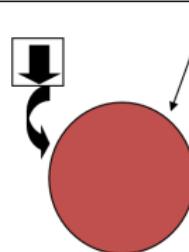
Objects are created dynamically using the `new` keyword.

```
// CircleTest.java: A test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle, bCircle;
        aCircle = new Circle(); //aCircle now points to an object
        bCircle = new Circle(); //bCircle now points to an object
    }
}
```

`aCircle = new Circle();`



`bCircle = new Circle();`

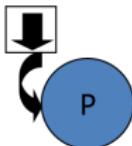


Assigning References of a Class

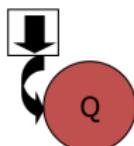
```
// CircleTest.java: A test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle, bCircle;
        aCircle = new Circle(); //aCircle now points to an object
        bCircle = new Circle(); //bCircle now points to an object
        bCircle = aCircle; // Assining a class reference
    }
}
```

Before Assignment

aCircle

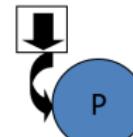


bCircle



After Assignment

aCircle



bCircle



Garbage Collection in Java

- In the previous example, object **Q** does not have a valid reference and, therefore, cannot be used in future.
- The object becomes a candidate for **Java Automatic Garbage Collection**.
 - ▶ Java automatically collects garbage periodically, and frees the memory of unused objects and makes this memory available for future use; you do not have to do this explicitly in the program.

Using Instance Variables and Methods

Syntax:

```
<objectName>.<varibaleName>;  
<objectName>.<methodName>(<arguments>);
```

Syntax is similar to C syntax for accessing data defined in a structure.

Example:

```
Circle aCircle = new Circle();  
double area;  
  
// Initialize centre and radius  
aCircle.centreX = 2.0;  
aCircle.centreY = 2.0;  
aCircle.radius = 1.0;  
  
//Invoking methods or sending a "message" to methods  
area = aCircle.computeArea();  
aCircle.resize(2.0);
```

Using the Circle Class - Example

```
// CircleTest.java - Test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle = new Circle();
        aCircle.centreX = 10.0;
        aCircle.centreY = 20.0;
        aCircle.radius = 5.0;
        System.out.println("Radius = " + aCircle.radius);
        System.out.println("Circum: = " + aCircle.computeCircumference());
        System.out.println("Area = " + aCircle.computeArea());
        aCircle.resize(2.0);
        System.out.println("Radius = " + aCircle.radius);
    }
}
```

Program Output:

```
Radius = 5.0
Circum: = 31.41592653589793
Area = 78.53981633974483
Radius = 10.0
```

Back to the main method

- A program in Java is just a class that has a `main` method.
- When you give a command to run a Java program, the run-time system invokes the `main` method.
- The `main` is a `void` method, as indicated by its heading:

```
public static void main(String[] args) {  
}
```

- `static` - it is still to come - please wait!

Updating and Accessing Instance Variables

Updating and Accessing Instance Variables

- Generally initialising/updating/accessing instance variables is done by defining specific methods for each purpose.
- These methods are called **Accessor/Mutator** methods or informally as **Getter/Setter** methods.
- Initialise/update an instance variable using:

```
aCircle.setX(10); // mutator method or setter
```

- Access an instance variable using:

```
aCircle.getX(); // accessor method or getter
```

- Usually IDEs such as IntelliJ, Eclipse IDE support automatic code generation for getters and setters.
- You will see better reasons for using getters and setters when we learn topics such as *information hiding, visibility control and privacy*.
So please be patient if you are not convinced as to why we are doing this!

The Circle Class with Getters and Setters

```
public class Circle {  
    public double centreX, centreY, radius;  
    public double getCentreX() {  
        return centreX;  
    }  
    public void setCentreX(double centreX) {  
        this.centreX = centreX;  
    }  
    public double getCentreY() {  
        return centreY;  
    }  
    public void setCentreY(double centreY) {  
        this.centreY = centreY;  
    }  
    public double getRadius() {  
        return radius;  
    }  
    public void setRadius(double radius) {  
        this.radius = radius;  
    } // The rest of the code as before go below  
}
```

Using the Circle Class with Getters and Setters

```
// CircleTest.java - Test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle = new Circle();
        aCircle.setCentreX(10.0);
        aCircle.setCentreY(20.0);
        aCircle.setRadius(5.0);
        System.out.println("Radius = " + aCircle.getRadius());
        System.out.println("Circum: = " + aCircle.computeCircumference());
        System.out.println("Area = " + aCircle.computeArea());
        aCircle.resize(2.0);
        System.out.println("Radius = " + aCircle.getRadius());
    }
}
```

Program Output:

```
Radius = 5.0
Circum: = 31.41592653589793
Area = 78.53981633974483
Radius = 10.0
```

Initializing Objects using Constructors

- When objects are created, the initial value of the instance variables are set to default values based on the data type.
- In the previous examples, we set the initial values using the mutator/setter methods.

```
// CircleTest.java - Test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle = new Circle();
        aCircle.setCentreX(10.0);
        aCircle.setCentreY(20.0);
        aCircle.setRadius(5.0);
    }
}
```

- What if we have 100 attributes to initialise?
- What if we have 100 objects to initialise?
- We need a better... **method**

Constructors

How does this actually work?

```
Circle aCircle = new Circle();
```

- The right hand side *invokes* (or calls) a class' *constructor*
- Constructors are **methods**
- Constructors are used to initialize objects
- Constructors have the same name as the class
- Constructors cannot return values
- A class can have **one or more** constructors, each with a different set of parameters (called overloading; we'll cover this later)

Keyword

Constructor: A method used to **create** and **initialise** an object.

Defining Constructors

```
public <ClassName>(<arguments>) {  
    <block of code to execute>  
}
```

Default Circle constructor:

```
public Circle() {  
    centreX = 10.0;  
    centreY = 10.0;  
    radius = 5.0;  
}
```

More useful Circle constructor:

```
public Circle(double newCentreX, double newCentreY, double newRadius) {  
    centreX = newCentreX;  
    centreY = newCentreY;  
    radius = newRadius;  
}
```

Using Constructors

Previous Code (without a Circle constructor):

```
Circle aCircle = new Circle();
```

aCircle = new Circle();



At creation time the center and radius are not defined.

aCircle will have a centre (0.0, 0.0) and radius 0.0 – default values for variables.

New Code (with Circle Constructors):

```
Circle aCircle = new Circle();
```

aCircle = new Circle();



At creation time the constructor with no arguments will be called.

aCircle will have a centre (10.0, 10.0) and radius 5.0 – default values for variables.

Constructor Test - Example

```
public class CircleConstructorTest {  
    public static void main(String args[]) {  
  
        Circle circle_1 = new Circle();  
        System.out.println("Defined circle_1 with centre (" +  
            circle_1.getCentreX() + ", " + circle_1.getCentreY() + "  
            and radius " + circle_1.getRadius());  
  
        Circle circle_2 = new Circle(10.0, 20.0, 12.2);  
        System.out.println("Defined circle_2 with centre (" +  
            circle_2.getCentreX() + ", " + circle_2.getCentreY() + "  
            and radius " + circle_2.getRadius());  
    }  
}
```

Program Output:

```
Defined circle_1 with centre (10.0, 10.0) and radius 5.0  
Defined circle_2 with centre (10.0, 20.0) and radius 12.2
```

The Circle class with more Constructors

```
public class Circle {  
    public double centreX, centreY, radius;  
    public Circle() {  
        centreX = 10.0;  
        centreY = 10.0;  
        radius = 5.0;  
    }  
    public Circle(double newCentreX, double newCentreY, double newRadius) {  
        centreX = newCentreX;  
        centreY = newCentreY;  
        radius = newRadius;  
    }  
    public Circle(double newCentreX, double newCentreY) {  
        centreX = newCentreX;  
        centreY = newCentreY;  
    }  
    public Circle(double newRadius) {  
        radius = newRadius;  
    }  
    // More code here  
}
```

Method Overloading

- Methods have the same name; are distinguished by their signature:
 - ▶ number of arguments
 - ▶ type of arguments
 - ▶ position of arguments
- Any method can be overloaded (Constructors or other methods).
- **Method Overloading:** This is a form of *polymorphism* (same method different behaviour).
- *Do not confuse with Method Overriding (coming up soon!).*

Method Overloading and Polymorphism

Keyword

Polymorphism: Ability to process objects differently depending on their data type or class.

Keyword

Method Overloading: Ability to define methods with the same name but with different signatures (argument types and/or numbers).

Pitfall: Constructors

Let us look at our previous definition of the Circle Constructor.

```
public Circle(double newCentreX, double newCentreY, double newRadius) {  
    centreX = newCentreX;  
    centreY = newCentreY;  
    radius = newRadius;  
}
```

But what if we did the following instead?

```
public Circle(double centreX, double centreY, double radius) {  
    centreX = centreX;  
    centreY = centreY;  
    radius = radius;  
}
```

How does the code differentiate the two variables?

Introducing the `this` Keyword

Keyword

`this`: A **reference** to the **calling object**, the object that owns/is executing the method.

```
public class Circle {  
    public double centreX, centreY, radius;  
  
    public Circle() {  
        this.centreX = 10.0;  
        this.centreY = 10.0;  
        this.radius = 5.0;  
    }  
  
    public Circle(double centreX, double centreY, double radius) {  
        this.centreX = centreX;  
        this.centreY = centreY;  
        this.radius = radius;  
    }  
    // More methods go here  
}
```

Static Attributes and Methods

Static Members

Keyword

Static Members: Methods and attributes that are not specific to any object of the class.

Keyword

Static Variable: A variable that is shared among all objects of the class; a single instance is shared among classes. Such an attribute is accessed using the class name.

Keyword

Static Method: A method that does not depend on (access or modify) any instance variables of the class. Such a method is invoked (called) using the class name.

Defining Static Variables

Static attribute are shared between objects (only one copy): e.g. count of the number of objects of the type that has been created.

Adding a static attribute, numCircles, to the Circle class.

```
// Circle.java
public class Circle {
    // static (class) variable - one instance for the Circle class, numCircles
    public static int numCircles = 0;
    public double centreX, centreY, radius;

    // Constructors and other methods
    public Circle(double x, double y, double r){
        centreX = x; centreY = y; radius = r;
        numCircles++;
    }
    // Other methods go here
}
```

Using Static Variables

Let us now write a class CountCircles to use the static variable.

```
// CountCircles.java
public class CountCircles {
    public static void main(String args[]) {
        Circle circleA = new Circle( 10.0, 12.0, 20.0);
        System.out.println("Number of Circles = " + Circle.numCircles );
        Circle circleB = new Circle( 5.0, 3.0, 10.0);
        System.out.println("Number of Circles = " + Circle.numCircles );
    }
}
```

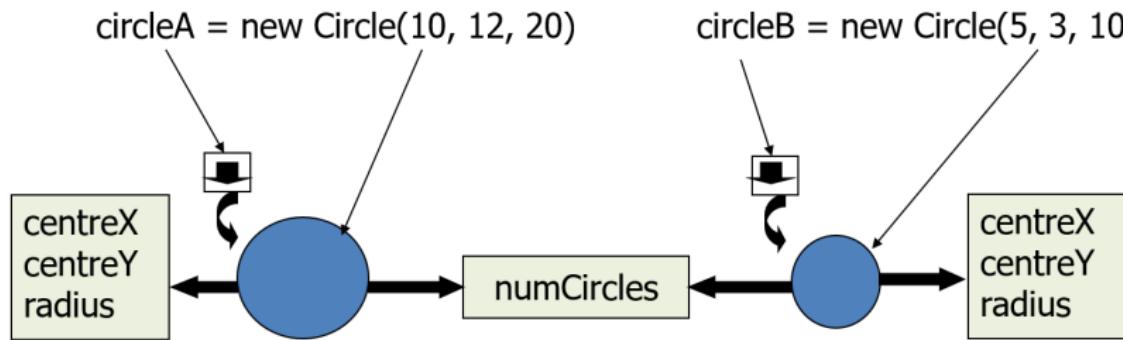
Program Output:

```
Number of Circles = 1
Number of Circles = 2
```

Instance vs Static Variables

Instance variables: One copy per object. e.g. centreX, centreY, radius (centre and radius in the circle)

Static variables: One copy per class. e.g. numCircles (total number of circle objects created)



Defining Static Methods

Adding a static method, `printNumCircles`, to the `Circle` class.

```
// Circle.java
public class Circle {
    // static (class) variable
    public static int numCircles = 0;
    public double centreX, centreY, radius;

    // Constructors and other methods
    public Circle(double x, double y, double r){
        centreX = x; centreY = y; radius = r;
        numCircles++;
    }

    // Static method to count the number of circles
    public static void printNumCircles() {
        System.out.println("Number of circles = " + numCircles);
    }

    // Other methods go here
}
```

Using Static Methods

Using the static method, `printNumCircles()`.

```
// CountCircles.java
public class CountCircles {
    public static void main(String args[]) {
        Circle circleA = new Circle( 10.0, 12.0, 20.0);
        Circle.printNumCircles();
        Circle circleB = new Circle( 5.0, 3.0, 10.0);
        Circle.printNumCircles();
    }
}
```

Program Output:

```
Number of Circles = 1
Number of Circles = 2
```

Using Static Methods

- Static methods can only call other static methods.
- Static methods can only access static data.
- Static methods cannot refer to Java keywords such as, `this` or `super` (will be introduced later) - because they are related to objects (class instances).
- Do not make all methods and attributes in your classes static; if you do that you may end up writing procedural programs using Java as opposed to good OO programs - you will be penalized for doing this in the assignments and exams.

Important: Before you decide to make an attribute or a method `static` think carefully - consider whether it is a class level member or an instance specific member.

Back to the main method

When a Java program is executed the Java virtual machine invokes the main method, which is a **static** method.

```
// HelloWorld.java: Display "Hello World!" on the screen
import java.lang.*;
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

Standard Methods in Java

Standard Methods

There are some methods, that are frequently used, that are provided as standard methods in every class.

We will look at three such methods:

- the `equals` method
- the `toString` method
- the copy constructor

Standard Methods - *equals*

```
public boolean equals(<ClassName> var) {  
    return <boolean expression>;  
}
```

- It is useful to be able to compare if two objects are **equal**
- Doing the equality test with `==` operator will only check if references are equal as opposed to checking if objects are equal
- How to determine if objects are equal is up to you; use **one or more** properties of the objects
- This is version one; we'll "improve" it as we go

Adding equals to Circle Class

We will now add the `equals` method to the `Circle` class.

How would you compare a `Circle` object to another `Circle` object?

```
public boolean equals(Circle circle) {  
    return Double.compare(circle.centreX, centreX) == 0 &&  
           Double.compare(circle.centreY, centreY) == 0 &&  
           Double.compare(circle.radius, radius) == 0;  
}
```

Standard Methods - `toString`

- What you if you want to print the attributes of the `Circle` class - is there an easy way?
- What would happen if you have:
 - ▶ `System.out.println(c_1);` - `c_1` is a reference to a `Circle` object
- The `toString` method which returns a `String` **representation** of an object is the way to go:
 - ▶ It is automatically called when the object is asked to act like a `String`, e.g. **printing** an object using: `System.out.println(c_1);`

```
public String toString() {  
    return <String>;  
}
```

Adding the `toString` method to the Circle Class

We will now add the `toString` method to the `Circle` class.

```
public class Circle {  
    // Other attributes and methods  
  
    public String toString() {  
        return "I am a Circle with {" + "centreX=" + centreX +  
               ", centreY=" + centreY +  
               ", radius=" + radius + '}';  
    }  
}
```

```
System.out.println(new Circle(5.0, 5.0, 40.0));
```

Program Output:

I am a Circle with {centreX=5.0, centreY=5.0, radius=40.0}

Standard Methods - *Copy Constructor*

```
public <ClassName>(<ClassName> var) {  
    <block of code to execute>  
}
```

- Is a constructor with a single argument of the same type as the class
- Creates a **separate copy** of the object sent as input
- The copy constructor should create an object that is a separate, independent object, but with the instance variables set so that it is an exact copy of the argument object
- In case some of the instance variables are references to other objects, a new object with the same state must be created using its copy constructor - *deep copy*

Adding a Copy Constructor to the Circle Class

```
public class Circle {  
    public double centreX, centreY, radius;  
    // Copy Constructor  
    Circle (Circle aCircle) {  
        if (aCircle == null) {  
            System.out.println("Fatal Error."); //Not a valid circle  
            System.exit(0);  
        }  
        this.centreX = aCircle.centreX;  
        this.centreY = aCircle.centreY;  
        this.radius = aCircle.radius;  
    }  
    // Other methods  
}
```

```
Circle c1 = new Circle(10.0, 10.0, 5.0); //a new object  
Circle c2 = c1; //a reference to the same object pointed by c1  
Circle c3 = new Circle(c1); //a new object - state is same as c1
```

Introducing Java Packages

Packages in Java

Keyword

Package: Allows to group classes and interfaces (will be introduced later) into bundles, that can then be handled together using an accepted naming convention.

Why would you group classes into packages?

- Works similar to libraries in C; can be developed, packaged, imported and used by other Java programs/classes.
- Allows reuse, rather than rewriting classes, you can use existing classes by importing them.
- Prevents naming conflicts.
 - ▶ Classes with the same name can be used in a program, uniquely identifying them by specifying the package they belong to.
- Allows access control - will learn more when we learn *Information Hiding/Visibility Control*.
- It is another level of **Encapsulation**.

Creating Java Packages

- To place a class in a package, the first statement in the Java class must be the `package` statement with the following syntax:

```
package <directory_name_1>.<directory_name_2>;
```

- This implies that the class in directory_2, which is a sub-directory of directory_1.

Example:

```
package utilities.shapes;  
  
public class Circle {  
    // Code for Circle goes here  
}
```

- `Circle.class` must be in directory `shapes`, which is a sub-directory of directory `utilities`

Using Java Packages

- To use classes in a package, the `import` statement, which can take one of the following forms must be used:

```
import <packageName>.*; // Imports all classes in the package  
import <packageName>.<className>; // Imports the particular class
```

- Once imported the, the class importing the package, can use the class.
- The parent directory where the classes are placed must be in the CLASSPATH environment variable - similar to PATH variable.

Example:

```
import utilities.shapes.Circle;  
public class CircleTest {  
    public static void main(String aargs[]) {  
        Circle my_circle = new Circle();  
    }  
}
```

- The parent directory of `utilities` directory, must be in the CLASSPATH environment variable.

The Default Package

- All the classes in the current directory belong to an unnamed package called the **default** package - no package statement is needed.
- As long as the current directory (.) is part of the CLASSPATH variable, all the classes in the default package are automatically available to a program.
- If the CLASSPATH variable is set, the current directory must be included as one of the alternatives; otherwise, Java may not even be able to find the .class files for the program itself.
- If the CLASSPATH variable is not set, then all the class files for a program must be put in the current directory.

This was a very brief introduction to packages, hence will not be assessed in this subject; if you want to use packages you will have to read up more. Here is one good [link](#).

Information Hiding

Information Hiding

- The OO design paradigm allows information related to classes/objects (i.e. attributes and methods) to be grouped together - **Encapsulation**.
- Actions on objects can be performed through methods of the class **interface** to the class.
- The OO design paradigm also supports **Information Hiding**; some attributes and methods can be hidden from the user.
- Information Hiding is also referred to a **Visibility Control**.

Keyword

Information Hiding: Ability to “hide” the details of a class from the outside world.

Keyword

Access Control: Preventing an outside class from **manipulating** the properties of another class in **undesired** ways.

Information Hiding

Java provides control over the **visibility (access)** of variables and methods through **visibility modifiers**:

- This allows to safely seal data within the capsule of the class
- Prevents programmers from relying on details of class implementation
- Helps in protecting against accidental or wrong usage
- Keeps code elegant and clean (easier to maintain)
- Enables to provide access to the object through a clean interface

Visibility Modifiers

Keyword

public: Keyword when applied to a class, method or attribute makes it available/visible everywhere (within the class and outside the class).

Keyword

private: Keyword when applied to a method or attribute of a class, makes them only visible within that class. Private methods and attributes are not visible within *subclasses*, and are not inherited.

Keyword

protected: Keyword when applied to a method or attribute of a class, makes them only visible within that class, *subclasses* and also within all classes that are in the same package as that class. They are also visible to *subclasses* in other packages.

Note: We will learn about *subclasses* when we learn Inheritance.

Visibility Modifiers

| Modifier | Class | Package | Subclass | Outside |
|------------------------|--------------|----------------|-----------------|----------------|
| <code>public</code> | Y | Y | Y | Y |
| <code>protected</code> | Y | Y | Y | N |
| <code>default</code> | Y | Y | N | N |
| <code>private</code> | Y | N | N | N |

The Circle Class with Visibility Modifiers

- Attributes of the class must be made **private** and accessed through getter/setter methods, which are **public**.
- Methods that other classes do not call must be defined as **private**.

```
public class Circle {  
    private double centreX, centreY, radius;  
  
    //Methods to get and set the instance variables  
    public double getX() { return centreX; }  
    public double getY() { return centreY; }  
    public double getR() { return radius; }  
    public void setX(double centreX) { this.centreX = centreX; }  
    public void setY(double centreY) { this.centreY = centreY; }  
    public void setR(double radius) { this.radius = radius; }  
    // Other methods  
}
```

Mutability

Keyword

Mutable: A class that contains public mutator methods or other public methods that can change the instance variables is called a *mutable class*, and its objects are called *mutable objects*.

Keyword

Immutable: A class that contains no methods (other than constructors) that change any of the instance variables is called an *immutable class*, and its objects are called *immutable objects*.

Back to the Circle Class

```
// Circle.java
public class Circle {
    private double centreX, centreY, radius;
    private static int numCircles;

    public Circle(double newCentreX, double newCentreY, double newRadius) {
        public double getCentreX() {.. }
        public void setCentreX(double centreX) {.. }
        public double getCentreY() {.. }
        public void setCentreY(double centreY) {.. }
        public double getRadius() {..}
        public void setRadius(double radius) {..}
        public double computeCircumference() {..}
        public double computeArea() {..}
        public void resize(double factor) {..}
        public static int getNumCircles() {..}
    }
}
```

Is this an immutable class?

How would you create an immutable Circle class?

Creating an Immutable Class

```
// ImmutableCircle.java
public class ImmutableCircle {
    private final double centreX, centreY, radius;
    private static int numCircles;

    public ImmutableCircle(double newCentreX, double newCentreY,
                          double newRadius) {..}

    public double getCentreX() {..}
    public double getCentreY() {..}
    public double getRadius() {..}
    public double computeCircumference () {..}
    public double computeArea () {..}
    public static int getNumCircles() {..}
}
```

Delegation through Association

Delegation

- A class can **delegate** its responsibilities to other classes.
- An object can **invoke methods** in other objects through **containership**.
- This is an **Association** relationship between the classes (will be explained in more detail later).

Delegation - Example

We will demonstrate the Association relationship and Delegation through a Point class contained within the Circle class.

```
public class Point {  
    private double xCoord;  
    private double yCoord;  
  
    // Constructor  
    ....  
  
    public double getXCoord() {  
        return xCoord;  
    }  
  
    public double getYCoord() {  
        return yCoord;  
    }  
}
```

Delegation - Example

```
public class Circle {  
    private Point centre;  
    private double radius;  
  
    public Circle(Point centre, double radius) {  
        this.centre = centre;  
        this.radius = radius;  
    }  
    public double getX() {  
        return centre.getXCoord();  
    }  
    public double getY() {  
        return centre.getYCoord();  
    }  
    // Other methods go here  
}
```

A Point object is contained in the Circle object; methods in a Circle object can call methods in the Point object using the reference to the object, centre.

Wrapper Classes

Back to Primitive Data Types

Primitives like `int` and `double`:

- Contain **only** data
- Do **not** have attributes or methods
- Can't "perform actions" like parsing

Keyword

Primitive: A unit of information that contains only data, and has no attributes or methods

Wrapper Classes

- Java provides “wrapper” classes for primitives
- Allows primitive data types to be “packaged” or “boxed” into objects
- Allows primitives to “pretend” that they are classes (this is important later)
- **Provides extra functionality for primitives**

Keyword

Wrapper: A class that gives extra functionality to primitives like `int`, and lets them behave like objects

Wrapper Classes

| Primitive | Wrapper Class |
|-----------|---------------|
| boolean | Boolean |
| byte | Byte |
| char | Character |
| int | Integer |
| float | Float |
| double | Double |
| long | Long |
| short | Short |

Integer Class

Provides a number of methods such as:

- Reverse: Integer.reverse(10)
- Rotate Left: Integer.rotateLeft(10, 2)
- Signum: Integer.signum(-10)
- Parsing: Integer.parseInt("10")

```
Integer x = Integer.parseInt("20");
int y = x;
Integer z = 2*x;
```

Parsing

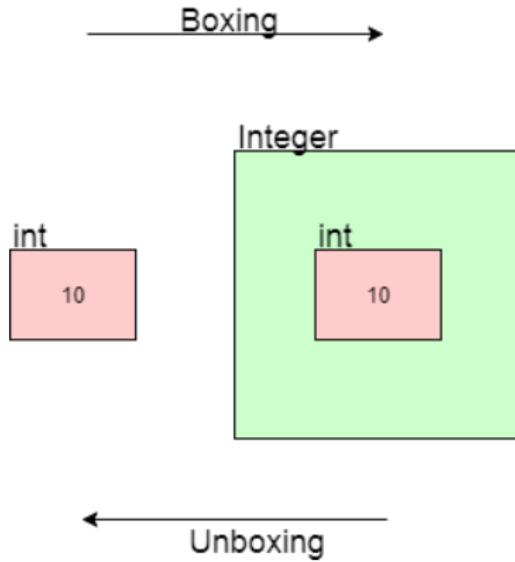
Every wrapper class has a parse function:

- `xxx var = XXX.parseXXX(<string>);`
- `int i = Integer.parseInt("1");`
- `double d = Double.parseDouble("1");`
- `boolean b = Boolean.parseBoolean("True");`

Keyword

Parsing: Processing one data type into another

Automatic Boxing/Unboxing



Keyword

(Un)Boxing: The process of converting a primitive to/from its equivalent wrapper class

Learning Outcomes

Upon completion of this topic you will be able to:

- Explain the difference between a *class* and an *object*
- Define classes, and give them *properties* and *behaviours*
- Implement and use classes
- Identify a series of well-defined classes from a *specification*
- Explain object oriented concepts: abstraction, encapsulation, information hiding and delegation
- Understand the role of Wrapper classes

References

- Absolute Java by Walter Savitch (Fourth Edition), Chapters 4 & 5

SWEN20003
Object Oriented Software Development

Classes and Objects

Semester 1, 2020

The Road So Far

Lectures

- Subject Introduction
- A Quick Tour of Java

Learning Outcomes

Upon completion of this topic you will be able to:

- Explain the difference between a *class* and an *object*
- Define classes, and give them *properties* and *behaviours*
- Implement and use classes
- Identify a series of well-defined classes from a *specification*
- Explain object oriented concepts: abstraction, encapsulation, information hiding and delegation
- Understand the role of Wrapper classes

Introduction

All programming languages support four basic concepts:

- Calculation: constants, variables, operators, expressions
- Selection: `if-else`, `switch`, ?
- Iteration: `while`, `do`, `for`
- **Abstraction:** The process of creating self-contained units of software that allows the solution to be parameterized and therefore more general purpose

Abstraction is the fundamental concept that differentiates procedural programming languages such as C from Object Oriented languages such as Java, C++.

Abstraction in Procedural Languages

Abstraction in procedural languages is provided through *functions* or *procedures*.

Functions manipulate external data to by performing operations on them.

Example of a function in C that calculates the average of two floating point numbers:

```
float calculate_average (float a, float b) {  
    float result;  
    result = (a + b)/2;  
    return result;  
}
```

Abstraction in Object Oriented Languages

Abstraction in Object Oriented (OO) languages is provided through an *Abstract Data Type (ADT)*, which contains *data* and *functions* that operate on data.

In Java a **Class** is an implementation on an **Abstract Data Type**.

Classes

- A “generalization” of a real world (or “problem world”) entity
 - ▶ A physical real world thing, like a student or book
 - ▶ An abstract real world thing, like a university subject
 - ▶ An even more abstract thing like a list or a string (data)
- Represents a template for things that have common properties
- Contains *attributes* and *methods*
- Defines a new **data type**

Keyword

Class: Fundamental unit of abstraction in *Object Oriented Programming*.
Represents an “entity” that is part of a problem.

Objects

- Are an *instance* of a class
- Contain **state**, or dynamic information
- “**X** is of type A”, “**X** is an object of the class A”, and “**X** is an instance of the class A” are all equivalent

Keyword

Object: A specific, concrete example of a class

Keyword

Instance: An object that exists in your code

Motivating Example

Throughout this topic we will be referring to the following specification:

Develop a system (a set of classes) for a simple Drawing Pad application. The application should allow drawing different types of shapes, such as circles, squares, rectangles, display their geometrical properties: e.g. area, circumference. It should also allow different types of actions such as moving, resizing to the performed on shapes.

How would you develop this, right now? What additional information do you need?

Drawing Pad Application - Classes

What classes can we use for our example problem?

Fundamental:

- Drawing Pad
- Circle
- Square
- Other shapes

Additional:

- Drawing Tool
- Paint Brush
- Fill Colour
- Fill Type
- Many more

Identifying Attributes and Methods

Let us consider the `Circle` class.

Can you identify the *attributes* and *methods* of the class?

Attributes:

- Centre
- Radius
- Fill Colour, Fill Type
- Many more

Methods (Operations):

- Compute Circumference
- Compute Area
- Move
- Resize
- Many more

Object Oriented Features

Following are some key features of the object oriented design paradigm:

- Data Abstraction
- Encapsulation
- Information Hiding
- Delegation
- Inheritance
- Polymorphism

Data Abstraction

Keyword

Data Abstraction: The technique of creating new data types that are well suited to an application by defining new classes.

A class is a special kind of programmer-defined data type.

- For example, by creating classes such as Circle, Drawing Pad, you are creating new data types, that can be used in applications.

A class is somewhat close to a structure in C but has additional features - attributes and methods.

The class definition determines the types of data (attributes) that an object can contain, as well as the actions (methods) it can perform.

Encapsulation

Keyword

Encapsulation: The ability to group data (attributes) and methods that manipulate the data to a single entity though defining a class.

A class encapsulates data and the methods that operate on the data into a single unit.

This method of encapsulation is unique to OO programming and is not provided by the procedural programming paradigm.

Defining a Class

Defining a Class

Syntax:

```
<visibility modifier> class <ClassName> {  
    <attribute declarations>  
    <method declarations>  
}
```

A bare bone class:

```
// Circle.java - Circle class definition  
public class Circle {  
}
```

Defining a Class - Adding Attributes

Attribute Syntax:

```
<visibility modifier> <type> <variable name>;
```

Adding attributes (also called data, fields) to the Circle class.

```
// Circle.java - Circle class definition
public class Circle {
    public double centreX; //centre x coordinate
    public double centreY; //centre y coordinate
    public double radius; //radius
}
```

Defining a Class - Adding Attributes

The attributes added to the Circle class in the above example are referred to as *instance variables*.

- these attributes maintain the **state** of the object; i.e. by giving values to centreX, centreY, radius we define a Circle object with particular size and position.

Keyword

Instance Variable: A **property** or **attribute** that is unique to each *instance* (*object*) of a class, which maintains the **state** of the object.

Defining a Class - Adding Methods

Method Syntax:

```
<visibility modifier> <void or typeReturned> myMethod(paramList)
{
    variable declarations
    statements
}
```

- If the method returns data, the data type must be specified in the method definition, otherwise, it is defined as **void**.
- If the method returns data, the method body must contain a return statement, which returns a variable of the specified return type.
- Variables can be declared inside the method - such variables are called *local variables*.

Note: Local variables are inside the method as opposed to the instance variables (introduced earlier) which are outside the method declaration.

Defining a Class - Adding Methods

Adding methods to Circle class.

```
// Circle.java
public class Circle {
    public double centreX;
    public double centreY;
    public double radius;

    public double computeCircumference () {
        double circum = 2 * Math.PI * radius;
        return circum;
    }
    public double computeArea () {
        double area = Math.PI * radius * radius;
        return area;
    }
    public void resize (double factor) {
        radius = radius * factor;
    }
}
```

Using a Class

Using the Circle class

Follow the steps below to use the `Circle` class we just created.

- Create a file `Circle.java` and write the code.
Note: the file name should match the class name.
- You can use an Integrated Development Environment (IDE), such as IntelliJ for this (will be introduced in the workshops), but in this instance you can use a text editor such as notepad, wordpad, vim, kate etc.
- Compile the class using the following command:

```
javac Circle.java
```

This creates a file `Circle.class`

- `Circle` becomes a derived data type that can be used in a Java program.

Using the Circle class

By creating the Circle class, you have created a new data type **Circle - Data Abstraction**.

- Variables of type Circle can be now defined in a program
- Circle is a **Derived Data Type** (as opposed to a Primitive Data Type such as int, float)

Example:

```
/* CircleTest.java: A test program to test the Circle class */
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle;
        Circle bCircle
    }
}
```

Using the Circle class

The declarations:

```
Circle aCircle;  
Circle bCircle;
```

in the previous example did not create Circle objects.

aCircle and bCircle are simply **references** to Circle objects (not objects):

- Currently they point to nothing, hence **null references**.

aCircle



bCircle



Points to nothing
(Null Reference)

Points to nothing
(Null Reference)

The `null` Reference

Keyword

null: The Java keyword for “no object here”. Null objects can’t be “accessed” to get variables or methods, or used in any way.

Instantiating a Class

Objects are **null** until they are *instantiated*.

Keyword

Instantiate: To create an object of a class

```
// Instantiate an Circle object  
Circle circle_1 = new Circle();
```

Keyword

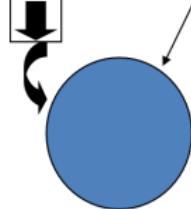
new: Directs the JVM to allocate memory for an object, or *instantiate* it

Creating Objects

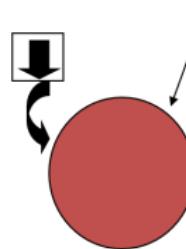
Objects are created dynamically using the `new` keyword.

```
// CircleTest.java: A test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle, bCircle;
        aCircle = new Circle(); //aCircle now points to an object
        bCircle = new Circle(); //bCircle now points to an object
    }
}
```

`aCircle = new Circle();`



`bCircle = new Circle();`

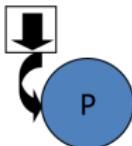


Assigning References of a Class

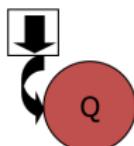
```
// CircleTest.java: A test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle, bCircle;
        aCircle = new Circle(); //aCircle now points to an object
        bCircle = new Circle(); //bCircle now points to an object
        bCircle = aCircle; // Assining a class reference
    }
}
```

Before Assignment

aCircle

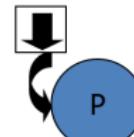


bCircle



After Assignment

aCircle



bCircle



Garbage Collection in Java

- In the previous example, object **Q** does not have a valid reference and, therefore, cannot be used in future.
- The object becomes a candidate for **Java Automatic Garbage Collection**.
 - ▶ Java automatically collects garbage periodically, and frees the memory of unused objects and makes this memory available for future use; you do not have to do this explicitly in the program.

Using Instance Variables and Methods

Syntax:

```
<objectName>.<varibaleName>;  
<objectName>.<methodName>(<arguments>);
```

Syntax is similar to C syntax for accessing data defined in a structure.

Example:

```
Circle aCircle = new Circle();  
double area;  
  
// Initialize centre and radius  
aCircle.centreX = 2.0;  
aCircle.centreY = 2.0;  
aCircle.radius = 1.0;  
  
//Invoking methods or sending a "message" to methods  
area = aCircle.computeArea();  
aCircle.resize(2.0);
```

Using the Circle Class - Example

```
// CircleTest.java - Test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle = new Circle();
        aCircle.centreX = 10.0;
        aCircle.centreY = 20.0;
        aCircle.radius = 5.0;
        System.out.println("Radius = " + aCircle.radius);
        System.out.println("Circum: = " + aCircle.computeCircumference());
        System.out.println("Area = " + aCircle.computeArea());
        aCircle.resize(2.0);
        System.out.println("Radius = " + aCircle.radius);
    }
}
```

Program Output:

```
Radius = 5.0
Circum: = 31.41592653589793
Area = 78.53981633974483
Radius = 10.0
```

Back to the main method

- A program in Java is just a class that has a `main` method.
- When you give a command to run a Java program, the run-time system invokes the `main` method.
- The `main` is a `void` method, as indicated by its heading:

```
public static void main(String[] args) {  
}
```

- `static` - it is still to come - please wait!

Updating and Accessing Instance Variables

Updating and Accessing Instance Variables

- Generally initialising/updating/accessing instance variables is done by defining specific methods for each purpose.
- These methods are called **Accessor/Mutator** methods or informally as **Getter/Setter** methods.
- Initialise/update an instance variable using:

```
aCircle.setX(10); // mutator method or setter
```

- Access an instance variable using:

```
aCircle.getX(); // accessor method or getter
```

- Usually IDEs such as IntelliJ, Eclipse support automatic code generation for getters and setters.
- You will see better reasons for using getters and setters when we learn topics such as *information hiding, visibility control and privacy*. So please be patient if you are not convinced as to why we are doing this!

The Circle Class with Getters and Setters

```
public class Circle {  
    public double centreX, centreY, radius;  
    public double getCentreX() {  
        return centreX;  
    }  
    public void setCentreX(double centreX) {  
        this.centreX = centreX;  
    }  
    public double getCentreY() {  
        return centreY;  
    }  
    public void setCentreY(double centreY) {  
        this.centreY = centreY;  
    }  
    public double getRadius() {  
        return radius;  
    }  
    public void setRadius(double radius) {  
        this.radius = radius;  
    } // The rest of the code as before go below  
}
```

Using the Circle Class with Getters and Setters

```
// CircleTest.java - Test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle = new Circle();
        aCircle.setCentreX(10.0);
        aCircle.setCentreY(20.0);
        aCircle.setRadius(5.0);
        System.out.println("Radius = " + aCircle.getRadius());
        System.out.println("Circum: = " + aCircle.computeCircumference());
        System.out.println("Area = " + aCircle.computeArea());
        aCircle.resize(2.0);
        System.out.println("Radius = " + aCircle.getRadius());
    }
}
```

Program Output:

```
Radius = 5.0
Circum: = 31.41592653589793
Area = 78.53981633974483
Radius = 10.0
```

Initializing Objects using Constructors

- When objects are created, the initial value of the instance variables are set to default values based on the data type.
- In the previous examples, we set the initial values using the mutator/setter methods.

```
// CircleTest.java - Test program to test the Circle class
public class CircleTest {
    public static void main(String args[]) {
        Circle aCircle = new Circle();
        aCircle.setCentreX(10.0);
        aCircle.setCentreY(20.0);
        aCircle.setRadius(5.0);
    }
}
```

- What if we have 100 attributes to initialise?
- What if we have 100 objects to initialise?
- We need a better... **method**

Constructors

How does this actually work?

```
Circle aCircle = new Circle();
```

- The right hand side *invokes* (or calls) a class' *constructor*
- Constructors are **methods**
- Constructors are used to initialize objects
- Constructors have the same name as the class
- Constructors cannot return values
- A class can have **one or more** constructors, each with a different set of parameters (called overloading; we'll cover this later)

Keyword

Constructor: A method used to **create** and **initialise** an object.

Defining Constructors

```
public <ClassName>(<arguments>) {  
    <block of code to execute>  
}
```

Default Circle constructor:

```
public Circle() {  
    centreX = 10.0;  
    centreY = 10.0;  
    radius = 5.0;  
}
```

More useful Circle constructor:

```
public Circle(double newCentreX, double newCentreY, double newRadius) {  
    centreX = newCentreX;  
    centreY = newCentreY;  
    radius = newRadius;  
}
```

Using Constructors

Previous Code (without a Circle constructor):

```
Circle aCircle = new Circle();
```

aCircle = new Circle();



At creation time the center and radius are not defined.

aCircle will have a centre (0.0, 0.0) and radius 0.0 – default values for variables.

New Code (with Circle Constructors):

```
Circle aCircle = new Circle();
```

aCircle = new Circle();



At creation time the constructor with no arguments will be called.

aCircle will have a centre (10.0, 10.0) and radius 5.0 – default values for variables.

Constructor Test - Example

```
public class CircleConstructorTest {  
    public static void main(String args[]) {  
  
        Circle circle_1 = new Circle();  
        System.out.println("Defined circle_1 with centre (" +  
            circle_1.getCentreX() + ", " + circle_1.getCentreY() + "  
            and radius " + circle_1.getRadius());  
  
        Circle circle_2 = new Circle(10.0, 20.0, 12.2);  
        System.out.println("Defined circle_2 with centre (" +  
            circle_2.getCentreX() + ", " + circle_2.getCentreY() + "  
            and radius " + circle_2.getRadius());  
    }  
}
```

Program Output:

```
Defined circle_1 with centre (10.0, 10.0) and radius 5.0  
Defined circle_2 with centre (10.0, 20.0) and radius 12.2
```

The Circle class with more Constructors

```
public class Circle {  
    public double centreX, centreY, radius;  
    public Circle() {  
        centreX = 10.0;  
        centreY = 10.0;  
        radius = 5.0;  
    }  
    public Circle(double newCentreX, double newCentreY, double newRadius) {  
        centreX = newCentreX;  
        centreY = newCentreY;  
        radius = newRadius;  
    }  
    public Circle(double newCentreX, double newCentreY) {  
        centreX = newCentreX;  
        centreY = newCentreY;  
    }  
    public Circle(double newRadius) {  
        radius = newRadius;  
    }  
    // More code here  
}
```

Method Overloading

- Methods have the same name; are distinguished by their signature:
 - ▶ number of arguments
 - ▶ type of arguments
 - ▶ position of arguments
- Any method can be overloaded (Constructors or other methods).
- **Method Overloading:** This is a form of *polymorphism* (same method different behaviour).
- *Do not confuse with Method Overriding (coming up soon!).*

Method Overloading and Polymorphism

Keyword

Polymorphism: Ability to process objects differently depending on their data type or class.

Keyword

Method Overloading: Ability to define methods with the same name but with different signatures (argument types and/or numbers).

Pitfall: Constructors

Let us look at our previous definition of the Circle Constructor.

```
public Circle(double newCentreX, double newCentreY, double newRadius) {  
    centreX = newCentreX;  
    centreY = newCentreY;  
    radius = newRadius;  
}
```

But what if we did the following instead?

```
public Circle(double centreX, double centreY, double radius) {  
    centreX = centreX;  
    centreY = centreY;  
    radius = radius;  
}
```

How does the code differentiate the two variables?

Introducing the `this` Keyword

Keyword

`this`: A **reference** to the **calling object**, the object that owns/is executing the method.

```
public class Circle {  
    public double centreX, centreY, radius;  
  
    public Circle() {  
        this.centreX = 10.0;  
        this.centreY = 10.0;  
        this.radius = 5.0;  
    }  
  
    public Circle(double centreX, double centreY, double radius) {  
        this.centreX = centreX;  
        this.centreY = centreY;  
        this.radius = radius;  
    }  
    // More methods go here  
}
```

Static Attributes and Methods

Static Members

Keyword

Static Members: Methods and attributes that are not specific to any object of the class.

Keyword

Static Variable: A variable that is shared among all objects of the class; a single instance is shared among classes. Such an attribute is accessed using the class name.

Keyword

Static Method: A method that does not depend on (access or modify) any instance variables of the class. Such a method is invoked (called) using the class name.

Defining Static Variables

Static attribute are shared between objects (only one copy): e.g. count of the number of objects of the type that has been created.

Adding a static attribute, numCircles, to the Circle class.

```
// Circle.java
public class Circle {
    // static (class) variable - one instance for the Circle class, num
    public static int numCircles = 0;
    public double centreX, centreY, radius;

    // Constructors and other methods
    public Circle(double x, double y, double r){
        centreX = x; centreY = y; radius = r;
        numCircles++;
    }
    // Other methods go here
}
```

Using Static Variables

Let us now write a class CountCircles to use the static variable.

```
// CountCircles.java
public class CountCircles {
    public static void main(String args[]) {
        Circle circleA = new Circle( 10.0, 12.0, 20.0);
        System.out.println("Number of Circles = " + Circle.numCircles );
        Circle circleB = new Circle( 5.0, 3.0, 10.0);
        System.out.println("Number of Circles = " + Circle.numCircles );
    }
}
```

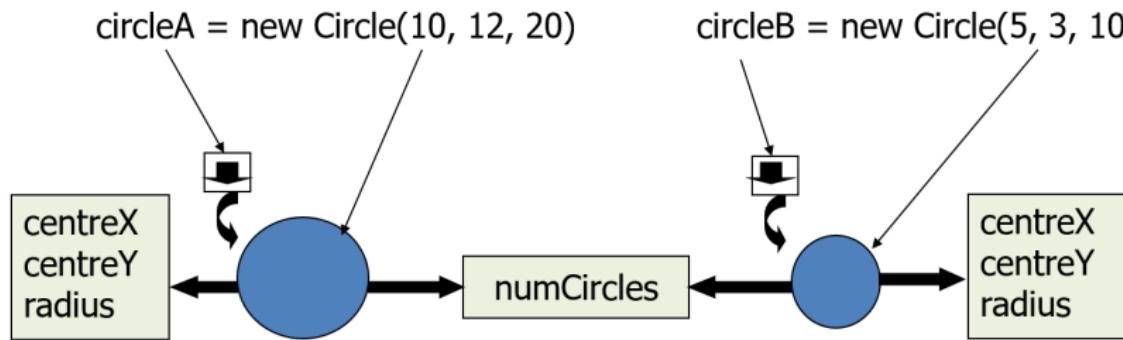
Program Output:

```
Number of Circles = 1
Number of Circles = 2
```

Instance vs Static Variables

Instance variables: One copy per object. e.g. centreX, centreY, radius (centre and radius in the circle)

Static variables: One copy per class. e.g. numCircles (total number of circle objects created)



Defining Static Methods

Adding a static method, `printNumCircles`, to the `Circle` class.

```
// Circle.java
public class Circle {
    // static (class) variable
    public static int numCircles = 0;
    public double centreX, centreY, radius;

    // Constructors and other methods
    public Circle(double x, double y, double r){
        centreX = x; centreY = y; radius = r;
        numCircles++;
    }

    // Static method to count the number of circles
    public static void printNumCircles() {
        System.out.println("Number of circles = " + numCircles);
    }

    // Other methods go here
}
```

Using Static Methods

Using the static method, `printNumCircles()`.

```
// CountCircles.java
public class CountCircles {
    public static void main(String args[]) {
        Circle circleA = new Circle( 10.0, 12.0, 20.0);
        Circle.printNumCircles();
        Circle circleB = new Circle( 5.0, 3.0, 10.0);
        Circle.printNumCircles();
    }
}
```

Program Output:

```
Number of Circles = 1
Number of Circles = 2
```

Using Static Methods

- Static methods can only call other static methods.
- Static methods can only access static data.
- Static methods cannot refer to Java keywords such as, `this` or `super` (will be introduced later) - because they are related to objects (class instances).
- Do not make all methods and attributes in your classes static; if you do that you may end up writing procedural programs using Java as opposed to good OO programs - you will be penalized for doing this in the assignments and exams.

Important: Before you decide to make an attribute or a method `static` think carefully - consider whether it is a class level member or an instance specific member.

Back to the main method

When a Java program is executed the Java virtual machine invokes the main method, which is a **static** method.

```
// HelloWorld.java: Display "Hello World!" on the screen
import java.lang.*;
public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

Standard Methods in Java

Standard Methods

There are some methods, that are frequently used, that are provided as standard methods in every class.

We will look at three such methods:

- the `equals` method
- the `toString` method
- the copy constructor

Standard Methods - *equals*

- It is useful to be able to compare if two objects are **equal**
- Doing the equality test with `==` operator will only check if references are equal as opposed to checking if objects are equal
- The **equals** method is a class is used for checking if the objects are equal
- How to determine if objects are equal is up to you; use **one or more** properties of the objects
- This is version one; we'll "improve" it as we go

```
public boolean equals(<ClassName> var) {  
    return <boolean expression>;  
}
```

Adding equals to Circle Class

We will now add the `equals` method to the `Circle` class.

How would you compare a `Circle` object to another `Circle` object?

```
public boolean equals(Circle circle) {  
    return Double.compare(circle.centreX, centreX) == 0 &&  
           Double.compare(circle.centreY, centreY) == 0 &&  
           Double.compare(circle.radius, radius) == 0;  
}
```

Standard Methods - `toString`

- What you if you want to print the attributes of the Circle class - is there an easy way?
- What would happen if you have:
 - ▶ `System.out.println(c_1);` - `c_1` is a reference to a Circle object
- The `toString` method which returns a String **representation** of an object is the way to go:
 - ▶ It is automatically called when the object is asked to act like a String, e.g. **printing** an object using: `System.out.println(c_1);`

```
public String toString() {  
    return <String>;  
}
```

Adding the `toString` method to the Circle Class

We will now add the `toString` method to the `Circle` class.

```
public class Circle {  
    // Other attributes and methods  
  
    public String toString() {  
        return "I am a Circle with {" + "centreX=" + centreX +  
               ", centreY=" + centreY +  
               ", radius=" + radius + '}';  
    }  
}
```

```
System.out.println(new Circle(5.0, 5.0, 40.0));
```

Program Output:

I am a Circle with {centreX=5.0, centreY=5.0, radius=40.0}

Standard Methods - *Copy Constructor*

```
public <ClassName>(<ClassName> var) {  
    <block of code to execute>  
}
```

- Is a constructor with a single argument of the same type as the class
- Creates a **separate copy** of the object sent as input
- The copy constructor should create an object that is a separate, independent object, but with the instance variables set so that it is an exact copy of the argument object
- In case some of the instance variables are references to other objects, a new object with the same state must be created using its copy constructor - *deep copy*

Adding a Copy Constructor to the Circle Class

```
public class Circle {  
    public double centreX, centreY, radius;  
    // Copy Constructor  
    Circle (Circle aCircle) {  
        if (aCircle == null) {  
            System.out.println("Fatal Error."); //Not a valid circle  
            System.exit(0);  
        }  
        this.centreX = aCircle.centreX;  
        this.centreY = aCircle.centreY;  
        this.radius = aCircle.radius;  
    }  
    // Other methods  
}
```

```
Circle c1 = new Circle(10.0, 10.0, 5.0); //a new object  
Circle c2 = c1; //a reference to the same object pointed by c1  
Circle c3 = new Circle(c1); //a new object - state is same as c1
```

Classes and Objects - Road So Far

Lecture 1:

- Introduction to Classes and Objects
- Understanding Data Abstraction and Encapsulation
- Defining a Class
- Using a Class

Lecture 2:

- Updating and Accessing Instance Variables
- Initializing Objects using Constructors
- Static Attributes and Methods
- Standard Methods in Java

Lecture 3:

- Introducing Java Packages
- Information Hiding
- Delegation through Association
- Wrapper Classes

Introducing Java Packages

Packages in Java

Keyword

Package: Allows to group classes and interfaces (will be introduced later) into bundles, that can then be handled together using an accepted naming convention.

Why would you group classes into packages?

- Works similar to libraries in C; can be developed, packaged, imported and used by other Java programs/classes.
- Allows reuse, rather than rewriting classes, you can use existing classes by importing them.
- Prevents naming conflicts.
 - ▶ Classes with the same name can be used in a program, uniquely identifying them by specifying the package they belong to.
- Allows access control - will learn more when we learn *Information Hiding/Visibility Control*.
- It is another level of **Encapsulation**.

Creating Java Packages

- To place a class in a package, the first statement in the Java class must be the `package` statement with the following syntax:

```
package <directory_name_1>.<directory_name_2>;
```

- This implies that the class in directory_2, which is a sub-directory of directory_1.

Example:

```
package utilities.shapes;  
  
public class Circle {  
    // Code for Circle goes here  
}
```

- `Circle.class` must be in directory `shapes`, which is a sub-directory of directory `utilities`

Using Java Packages

- To use classes in a package, the `import` statement, which can take one of the following forms must be used:

```
import <packageName>.*; // Imports all classes in the package  
import <packageName>.<className>; // Imports the particular class
```

- Once imported the, the class importing the package, can use the class.
- The parent directory where the classes are placed must be in the CLASSPATH environment variable - similar to PATH variable.

Example:

```
import utilities.shapes.Circle;  
public class CircleTest {  
    public static void main(String aargs[]) {  
        Circle my_circle = new Circle();  
    }  
}
```

- The parent directory of `utilities` directory, must be in the CLASSPATH environment variable.

The Default Package

- All the classes in the current directory belong to an unnamed package called the **default** package - no package statement is needed.
- As long as the current directory (.) is part of the CLASSPATH variable, all the classes in the default package are automatically available to a program.
- If the CLASSPATH variable is set, the current directory must be included as one of the alternatives; otherwise, Java may not even be able to find the .class files for the program itself.
- If the CLASSPATH variable is not set, then all the class files for a program must be put in the current directory.

This was a very brief introduction to packages, hence will not be assessed in this subject; if you want to use packages you will have to read up more. Here is one good [link](#).

Information Hiding

Information Hiding

- The OO design paradigm allows information related to classes/objects (i.e. attributes and methods) to be grouped together - **Encapsulation**.
- Actions on objects can be performed through methods of the class **interface** to the class.
- The OO design paradigm also supports **Information Hiding**; some attributes and methods can be hidden from the user.
- Information Hiding is also referred to a **Visibility Control**.

Keyword

Information Hiding: Ability to “hide” the details of a class from the outside world.

Keyword

Access Control: Preventing an outside class from **manipulating** the properties of another class in **undesired** ways.

Information Hiding

Java provides control over the **visibility (access)** of variables and methods through **visibility modifiers**:

- This allows to safely seal data within the capsule of the class
- Prevents programmers from relying on details of class implementation
- Helps in protecting against accidental or wrong usage
- Keeps code elegant and clean (easier to maintain)
- Enables to provide access to the object through a clean interface

Visibility Modifiers

Keyword

public: Keyword when applied to a class, method or attribute makes it available/visible everywhere (within the class and outside the class).

Keyword

private: Keyword when applied to a method or attribute of a class, makes them only visible within that class. Private methods and attributes are not visible within *subclasses*, and are not inherited.

Keyword

protected: Keyword when applied to a method or attribute of a class, makes them only visible within that class, *subclasses* and also within all classes that are in the same package as that class. They are also visible to *subclasses* in other packages.

Note: We will learn about *subclasses* when we learn Inheritance.

Visibility Modifiers

| Modifier | Class | Package | Subclass | Outside |
|------------------------|--------------|----------------|-----------------|----------------|
| <code>public</code> | Y | Y | Y | Y |
| <code>protected</code> | Y | Y | Y | N |
| <code>default</code> | Y | Y | N | N |
| <code>private</code> | Y | N | N | N |

The Circle Class with Visibility Modifiers

- Attributes of the class must be made **private** and accessed through getter/setter methods, which are **public**.
- Methods that other classes do not call must be defined as **private**.

```
public class Circle {  
    private double centreX, centreY, radius;  
  
    //Methods to get and set the instance variables  
    public double getX() { return centreX; }  
    public double getY() { return centreY; }  
    public double getR() { return radius; }  
    public void setX(double centreX) { this.centreX = centreX; }  
    public void setY(double centreY) { this.centreY = centreY; }  
    public void setR(double radius) { this.radius = radius; }  
    // Other methods  
}
```

Mutability

Keyword

Mutable: A class that contains public mutator methods or other public methods that can change the instance variables is called a *mutable class*, and its objects are called *mutable objects*.

Keyword

Immutable: A class that contains no methods (other than constructors) that change any of the instance variables is called an *immutable class*, and its objects are called *immutable objects*.

Back to the Circle Class

```
// Circle.java
public class Circle {
    private double centreX, centreY, radius;
    private static int numCircles;

    public Circle(double newCentreX, double newCentreY, double newRadius) {
        public double getCentreX() {.. }
        public void setCentreX(double centreX) {.. }
        public double getCentreY() {.. }
        public void setCentreY(double centreY) {.. }
        public double getRadius() {..}
        public void setRadius(double radius) {..}
        public double computeCircumference() {..}
        public double computeArea() {..}
        public void resize(double factor) {..}
        public static int getNumCircles() {..}
    }
}
```

Is this an immutable class?

How would you create an immutable Circle class?

Creating an Immutable Class

```
// ImmutableCircle.java
public class ImmutableCircle {
    private final double centreX, centreY, radius;
    private static int numCircles;

    public ImmutableCircle(double newCentreX, double newCentreY,
                          double newRadius) {..}

    public double getCentreX() {..}
    public double getCentreY() {..}
    public double getRadius() {..}
    public double computeCircumference () {..}
    public double computeArea () {..}
    public static int getNumCircles() {..}
}
```

Delegation through Association

Delegation

- A class can **delegate** its responsibilities to other classes.
- An object can **invoke methods** in other objects through **containership**.
- This is an **Association** relationship between the classes (will be explained in more detail later).

Delegation - Example

We will demonstrate the Association relationship and Delegation through a Point class contained within the Circle class.

```
public class Point {  
    private double xCoord;  
    private double yCoord;  
  
    // Constructor  
    ....  
  
    public double getXCoord() {  
        return xCoord;  
    }  
  
    public double getYCoord() {  
        return yCoord;  
    }  
}
```

Delegation - Example

```
public class Circle {  
    private Point centre;  
    private double radius;  
  
    public Circle(Point centre, double radius) {  
        this.centre = centre;  
        this.radius = radius;  
    }  
    public double getX() {  
        return centre.getXCoord();  
    }  
    public double getY() {  
        return centre.getYCoord();  
    }  
    // Other methods go here  
}
```

A Point object is contained in the Circle object; methods in a Circle object can call methods in the Point object using the reference to the object, centre.

Wrapper Classes

Back to Primitive Data Types

Primitives like `int` and `double`:

- Contain **only** data
- Do **not** have attributes or methods
- Can't "perform actions" like parsing

Keyword

Primitive: A unit of information that contains only data, and has no attributes or methods

Wrapper Classes

- Java provides “wrapper” classes for primitives
- Allows primitive data types to be “packaged” or “boxed” into objects
- Allows primitives to “pretend” that they are classes (this is important later)
- **Provides extra functionality for primitives**

Keyword

Wrapper: A class that gives extra functionality to primitives like `int`, and lets them behave like objects

Wrapper Classes

| Primitive | Wrapper Class |
|-----------|---------------|
| boolean | Boolean |
| byte | Byte |
| char | Character |
| int | Integer |
| float | Float |
| double | Double |
| long | Long |
| short | Short |

Integer Class

Provides a number of methods such as:

- Reverse: Integer.reverse(10)
- Rotate Left: Integer.rotateLeft(10, 2)
- Signum: Integer.signum(-10)
- Parsing: Integer.parseInt("10")

```
Integer x = Integer.parseInt("20");
int y = x;
Integer z = 2*x;
```

Parsing

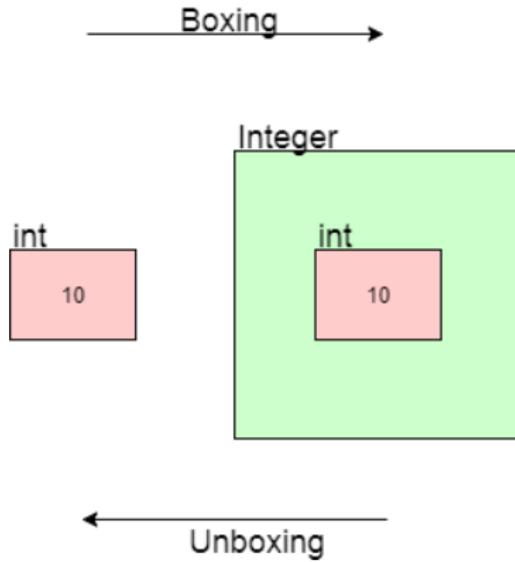
Every wrapper class has a parse function:

- `xxx var = XXX.parseXXX(<string>);`
- `int i = Integer.parseInt("1");`
- `double d = Double.parseDouble("1");`
- `boolean b = Boolean.parseBoolean("True");`

Keyword

Parsing: Processing one data type into another

Automatic Boxing/Unboxing



Keyword

(Un)Boxing: The process of converting a primitive to/from its equivalent wrapper class

Learning Outcomes

Upon completion of this topic you will be able to:

- Explain the difference between a *class* and an *object*
- Define classes, and give them *properties* and *behaviours*
- Implement and use classes
- Identify a series of well-defined classes from a *specification*
- Explain object oriented concepts: abstraction, encapsulation, information hiding and delegation
- Understand the role of Wrapper classes

References

- Absolute Java by Walter Savitch (Fourth Edition), Chapters 4 & 5