

SWEN20003

1 INTRO TO JAVA

COMPILING AND RUNNING

INPUTTING

COMMAND LINE ARGUMENTS

SCANNER

FILES

OUTPUT

PRINTING

FORMAT STRING

WRITING FILES

COMMENTING

IDENTIFIERS

PRIMITIVE DATA TYPES AND WRAPPER CLASSES

WRAPPER CLASSES

4 ARRAYS

4 STRINGS

CHARACTERS

CONDITIONALS

LOOPS

OBJECT ORIENTED FEATURES

2 CLASSES AND OBJECTS

OBJECTS

REFERENCES AND GARBAGE COLLECTION

STANDARD METHODS

JAVA PACKAGES

INFORMATION HIDING

VISIBILITY MODIFIER

DELEGATION THROUGH ASSOCIATION

SOFTWARE TOOLS AND BAGEL (NOT EXAMINABLE)

GIT

MAVEN

INTELLIJ DEBUGGER

[BAGEL](#)

[6 INHERITANCE AND POLYMORPHISM](#)

[SUPERCLASSES AND SUBCLASSES](#)

[UPCASTING AND DOWNCASTING](#)

[OVERRIDING AND OVERLOADING](#)

[RESTRICTING INHERITANCE](#)

[CHECKING OBJECT CLASSES](#)

[ABSTRACT CLASSES](#)

[ABSTRACT METHODS](#)

[TYPES OF INHERITANCE](#)

[INTERFACE](#)

[SORTING \(example\)](#)

[8 MODELLING CLASSES AND RELATIONSHIPS](#)

[UML](#)

[CLASS](#)

[RELATIONSHIPS](#)

[ASSOCIATION](#)

[GENERALISATION](#)

[REALISATION](#)

[DEPENDENCY](#)

[9 GENERICS](#)

[GENERIC CLASS](#)

[GENERIC METHOD](#)

[10 COLLECTIONS AND MAPS](#)

[11 DESIGN PATTERNS](#)

[12 EXCEPTIONS](#)

[13 SOFTWARE TESTING AND DESIGN](#)

[14 EVENT DRIVEN PROGRAMMING](#)

[SEQUENTIAL PROGRAMMING](#)

[EVENT-DRIVEN PROGRAMMING](#)

[SOFTWARE DEVELOPMENT FRAMEWORKS](#)

[15 ADVANCED JAVA CONCEPTS](#)

[Enumerated Types](#)

[Variadic Parameters](#)

[FUNCTIONAL INTERFACES AND LAMBDA EXPRESSIONS](#)

[LAMBDA EXPRESSIONS](#)

[METHOD REFERENCES](#)

[JAVA STREAMS](#)

1 INTRO TO JAVA

```
// HelloWorld.java
```

```
import java.lang.*
```

```
public class HelloWorld {
```

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

```
        ...
    }
```

```
}
```

```
// (opt.) imported by default
```

```
// class definition (must have same
// name as the file it is written in)
```

```
// the main method
```

```
// String args[] same as String[] args
```

COMPILING AND RUNNING

```
java FileName.java
```

```
compiling FileName.java
```

INPUTTING

COMMAND LINE ARGUMENTS

- information or data provided to a program when it is executed, accessible through the `args` variable

```
java FileName command line arguments
```

Entering Command Line Arguments & running *FileName.class*

- entering multi word strings: java *FileName* "command line" arguments
 - args[0] // command line
 - args[1] // arguments

Run configurations

```
args[0] // command
```

```
args[1] // line
```

```
args[2] // arguments
```

Accessing Command Line Arguments

- stored in *args* ([JAVA SYNTAX](#))

Disadvantage of Command Line Arguments

no interactivity

When should you use Command Line Arguments?

program configuration

- don't use in exams/tests unless explicitly told in the question

SCANNER

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

```
// in main method:
```

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

```
System.out.print("Enter an integer: ");
```

```
int n = scanner.nextInt();
```

```
System.out.println("You entered " + n);
```

```
scanner.close();
```

Getting input from user using Scanner class

```
// create an object of Scanner
```

```
// System.in: object representing standard input
// stream, or the command line/terminal (keyboard)
```

```
// take input from the user
```

```
// read an integer
```

	// close scanner object
String s = scanner.nextLine();	<p>reads a single line of text, up until a `return` or newline character</p> <ul style="list-style-type: none"> - continues from where we left off in the scanner (DOES NOT start on the next line) <pre> 1 import java.util.Scanner; 2 3 public class TestScanner3 { 4 public static void main(String[] args) { 5 Scanner scanner = new Scanner(System.in); 6 7 System.out.println("Enter your input: "); 8 double d = scanner.nextDouble(); 9 String s1 = scanner.nextLine(); 10 String s2 = scanner.nextLine(); 11 12 System.out.format("%3.2f , %s , %s", d, s1, s2); 13 } 14 } </pre> <p>Input: 5 6.7 7.2</p> <p>Output: 5.00, ,6.7</p> <ul style="list-style-type: none"> - add <code>scanner.nextLine()</code> to 'consume' the new line character after 5
boolean b = scanner.nextBoolean(); int i = scanner.nextInt(); double d = scanner.nextDouble(); String s2 = scanner.next();	<p>reads in a single values that matches the method name</p> <p>// reads in next <u>word</u></p>
What is the pitfall of nextXXX?	<pre> 1 import java.util.Scanner; 2 3 public class TestScanner2 { 4 public static void main(String[] args) { 5 Scanner scanner = new Scanner(System.in); 6 7 System.out.println("Enter your input: "); 8 double d = scanner.nextDouble(); 9 float f = scanner.nextFloat(); 10 int i = scanner.nextInt(); 11 12 System.out.format("%3.2f , %3.2f , %3d", d, f, i); 13 } 14 } </pre> <p>Input: 5 6.7 7.2</p> <p>Output: Error</p> <ul style="list-style-type: none"> - Scanner does not automatically downcast (eg. float → int) -
scanner.hasNext	Returns true if there is any input to be read
scanner.hasNextXXX	Returns true if the next `token` matches XXX
FILES	

<pre>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(); } } }</pre>	Using Scanner Class (for small files)
	libraries
<pre>import java.io.FileReader; import java.io.BufferedReader; import java.io.IOException; public class ReadFile1 { public static void main(String[] args) { try (BufferedReader br = new BufferedReader(new FileReader("test.txt"))) { String text = null; while ((text = br.readLine()) != null) { System.out.println(text); } } catch (Exception e) { e.printStackTrace(); } } }</pre>	Using BufferedReader <ul style="list-style-type: none"> - FileReader: low level file ("<i>filename.txt</i>") for simple character reading - BufferedReader: higher level file that permits reading Strings, not just characters - text: single line of text in file - DOES NOT compile without try and catch block
<p>// in while loop</p> <p>String cells[] = text.split(",");</p>	Parsing CSV <ul style="list-style-type: none"> - ! important for projects <pre>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", count, ingredient, cost*quantity); count++; } } catch (Exception e) { e.printStackTrace(); } } }</pre>

OUTPUT	
PRINTING	
System.out.print(<i>object</i>)	displays <i>object</i>
System.out.println(<i>object</i>)	displays <i>object</i> followed by a new line

System.out.printf(<i>format string</i> , ...)	
FORMAT STRING	
format string: % flags width precision conversion character	
Part	Explanation
flags	(default=right-justify)
	- left-justify
	+ output a plus (+) or minus (-) sign for a numerical value
	0 forces numerical values to be zero-padded
	, comma grouping separator (for numbers > 1000)
	<space> display a minus (-) sign if the number is negative or a space () if it is positive
width	(opt.) specifies the width for outputting the argument and represents the minimum number of characters to be written to the output. - includes commas and decimal points as characters
.precision	(opt.) specified the number of digits of precision when outputting a floating-point value or the length of a substring to extract from a String
conversion character	f float s string g 'optimal' float d integer c Unicode character
WRITING FILES	
<pre>import java.io.FileWriter; import java.io.PrintWriter; import java.io.IOException; public class FileWriter1 { 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 \$ \"iPhone\", 100000); } catch (IOException e) { e.printStackTrace(); } } }</pre>	using FileWriter <ul style="list-style-type: none"> - 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) - pw: file variable
pw.print pw.println pw.format	

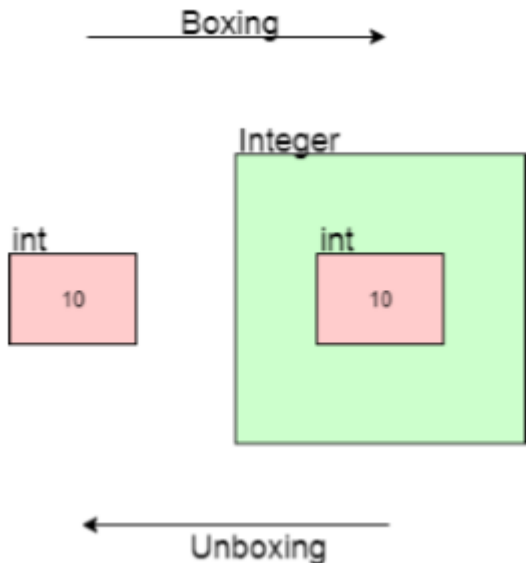
COMMENTING	
// single line comment	Single-line comment
/*	Multi-line comment

* multi- * line * comment */	
/** Documentation Comments */	Documentation comment

IDENTIFIERS							
- a name that uniquely identifies a program element such as a class, object, variable, method							
Identifier rules	<ul style="list-style-type: none"> - must not start with a digit - all characters must be letters, digits or underscore (_) symbol - case-sensitive - cannot be keywords/reserved words (public, class, void, ...) or predefined identifiers (System, String, ...) 						
Convention	<table> <tr> <td>Variables, Methods and Objects</td><td>camelCase</td></tr> <tr> <td>Classes</td><td>PascalCase</td></tr> <tr> <td>CONSTANT</td><td>UPPERCASE_SNAKE</td></tr> </table>	Variables, Methods and Objects	camelCase	Classes	PascalCase	CONSTANT	UPPERCASE_SNAKE
Variables, Methods and Objects	camelCase						
Classes	PascalCase						
CONSTANT	UPPERCASE_SNAKE						

PRIMITIVE DATA TYPES AND WRAPPER CLASSES									
<pre>int x = 2.99; // ❌ int y = (int)2.99; // ✅</pre>	<p>type casting</p> <p>Left can be assigned to anything on the right without type casting:</p> <ul style="list-style-type: none"> - byte → short → int → long → float → double - char → int <table> <tr> <td>✅</td><td>int x = 5;</td></tr> <tr> <td>❌</td><td>double y = x; // 5.0</td></tr> <tr> <td>❌</td><td>int z = y;</td></tr> <tr> <td>✅</td><td>int z = (int)y; // 5</td></tr> </table> <p>NOTE: boolean and int cannot be assigned to each other</p>	✅	int x = 5;	❌	double y = x; // 5.0	❌	int z = y;	✅	int z = (int)y; // 5
✅	int x = 5;								
❌	double y = x; // 5.0								
❌	int z = y;								
✅	int z = (int)y; // 5								

WRAPPER CLASSES								
primitive data types and Non-primitive Data Types	<table> <tr> <th>Primitive Data Type</th><th>Non-primitive Data Type (Wrapper Class)</th></tr> <tr> <td>starts with lowercase letter</td><td>starts with Uppercase letter</td></tr> <tr> <td>contains only data</td><td rowspan="2">can have attributes and methods</td></tr> <tr> <td>do not have attributes/methods</td></tr> </table>	Primitive Data Type	Non-primitive Data Type (Wrapper Class)	starts with lowercase letter	starts with Uppercase letter	contains only data	can have attributes and methods	do not have attributes/methods
Primitive Data Type	Non-primitive Data Type (Wrapper Class)							
starts with lowercase letter	starts with Uppercase letter							
contains only data	can have attributes and methods							
do not have attributes/methods								

	<table border="1"> <thead> <tr> <th>Primitive</th><th>Wrapper Class</th></tr> </thead> <tbody> <tr> <td>boolean</td><td>Boolean</td></tr> <tr> <td>byte</td><td>Byte</td></tr> <tr> <td>char</td><td>Character</td></tr> <tr> <td>int</td><td>Integer</td></tr> <tr> <td>float</td><td>Float</td></tr> <tr> <td>double</td><td>Double</td></tr> <tr> <td>long</td><td>Long</td></tr> <tr> <td>short</td><td>Short</td></tr> </tbody> </table>	Primitive	Wrapper Class	boolean	Boolean	byte	Byte	char	Character	int	Integer	float	Float	double	Double	long	Long	short	Short
Primitive	Wrapper Class																		
boolean	Boolean																		
byte	Byte																		
char	Character																		
int	Integer																		
float	Float																		
double	Double																		
long	Long																		
short	Short																		
<pre>Integer x = Integer.parseInt("20"); int y = x; Integer z = 2*x;</pre>	<pre>// unboxing // boxing</pre> 																		
<code>Wrapperclass.parseWrapper("");</code>	processing one data type into another																		
4 ARRAYS - mutable																			
<pre>basetype/Object[] varName; basetype/Object varName[];</pre>	array declaration - arrays are references - Object arrays - store references of instances of objects - default values = null																		
<pre>int[] intArray1 = {0, 1, 2, 3}; int[] intArray2 = new int[100];</pre>	array declaration and initialisation // initially initialised to 0 (because int array default)																		
	creating an array of objects																		
<pre>int[][] nums = new int[10][10]; for (int i=0; i<nums.length; i++) { nums[i] = new int[length_of_subarray] }</pre>	multi-dimensional array declaration and initialisation																		

array[0] array[-1] // give bounds error	indexing
array.length	length of array (final constant)
Arrays.equals(arr1, arr2)	comparing arrays - returns true if the element values are the same false otherwise
int[] arr = new int[5]; arr = new int[arr.length + 3];	resizing arrays - create new array with new length
Arrays.sort(arr1)	sorts array in ascending order
System.out.println(Arrays.toString(arr1))	printing array
for (Circle c : circleArray) { # code here # c is the current element in the array }	iterating through array -
array[index] = null	'remove' element from static array - there isn't a remove function .-
<p style="text-align: center;">4 STRINGS</p> <ul style="list-style-type: none"> - a class, data type - store a sequence of characters - must use double quotes (") - immutable Class <div style="border: 1px solid black; padding: 10px; margin: 10px 0;"> <pre>String s = "Hello World"; s.toUpperCase(); s.replace("e", "i"); s.substring(0, 2); s += " FIVE"; System.out.println(s);</pre> <p style="color: red; font-weight: bold; margin-top: 10px;">"Hello World FIVE"</p> </div>	
String s1 = "Hi" String s2 = "Hello World"	Declaring and Initialising Strings
\n \t \" \\	Special Characters - \ = escape character
System.out.println("1+1="+1+1) // Outputs 1+1=11 System.out.println("1+1="+ (1+1)) // Outputs 1+1=2	String Concatenation
s.length()	Returns the length of the string

s.toUpperCase()	Returns the string with all characters converted to uppercase								
s.toLowerCase	Returns the string with all characters converted to lowercase								
s.contains(substring)	Returns true if the given substring is contained within the calling <i>String</i>								
s.indexOf(substring)	Returns the index of the (first instance of the) given substring if it is contained within the calling <i>String</i> , and -1 otherwise								
s.substring(2, 7)	Returns the characters [arg1, arg2 - 1] () like s[2:7] in Python								
s1.equals(s2)	Checks for equality between any two class variables <ul style="list-style-type: none">- == will compare references								
s.matches(regex)									
s1.split(regex, limit)	Returns an array of strings computed by splitting the given string at the regex delimiter <ul style="list-style-type: none">- String <i>regex</i> - a delimiting regular expression- int <i>limit</i> - the resulting threshold<ul style="list-style-type: none">- >0: resulting array's length will not be more than <i>limit</i>- <0: resulting array can be of any size, pattern applied as many times as possible- =0: resulting array can be of any size, pattern applies as many times possible - trailing empty strings will be discarded								
String Equality									
	<table><tr><td>"HeLlLo" == "HeLlLo" →</td><td>true</td></tr><tr><td>String s = "Hello"; s == "HeLlLo" →</td><td>true</td></tr><tr><td>String s1 = "Hello"; String s2 = "Hello"; s1 == s2 →</td><td>true</td></tr><tr><td>String s = "Hello"; String s2 = new String ("Hello"); s1 == s2 →</td><td>false<ul style="list-style-type: none">- s and s2 are reference <u>objects</u>- must use .equals method</td></tr></table>	"HeLlLo" == "HeLlLo" →	true	String s = "Hello"; s == "HeLlLo" →	true	String s1 = "Hello"; String s2 = "Hello"; s1 == s2 →	true	String s = "Hello"; String s2 = new String ("Hello"); s1 == s2 →	false <ul style="list-style-type: none">- s and s2 are reference <u>objects</u>- must use .equals method
	"HeLlLo" == "HeLlLo" →	true							
	String s = "Hello"; s == "HeLlLo" →	true							
	String s1 = "Hello"; String s2 = "Hello"; s1 == s2 →	true							
String s = "Hello"; String s2 = new String ("Hello"); s1 == s2 →	false <ul style="list-style-type: none">- s and s2 are reference <u>objects</u>- must use .equals method								
CHARACTERS									
Character.isLetter(c)	checks if character is a letter (A-z)								
Character.isNumber(c)	checks if character is a number (0-9)								
Character.isLetterOrNumber(c)	returns true if a character is a letter (A-z) or number (0-9)								

--	--

CONDITIONALS	
<pre>if (condition) { ... } else if (condition) { ... } else { ... }</pre>	If Statement
<pre>switch (variable) { case val1: ... break; case val2: ... break; ... default: ... }</pre>	Switch Statement (can have multiple statements within the case; yea, it uses the break to know when the next case will start i guess - instead of brackets)

LOOPS	
<pre>for (condition) { ... }</pre>	For Loop
<pre>while (condition) { ... }</pre>	While Loop
<pre>do { ... } while (condition);</pre>	Do While Loop
<pre>loop1: while (...) { ... loop2: while (...) { ... break loop1; } }</pre>	you can give loops names (applicable for For, While and Do While loops)
continue;	skips the rest of the statements in the loop and repeats the loop
break <i>loopName</i> ;	breaks loop with <i>loopName</i> - if loopName not given, breaks inner loop

OBJECT ORIENTED FEATURES	
Abstraction	via <u>Abstract Data Types (ATD)</u>

	<ul style="list-style-type: none"> - classes are ATDs - versus procedural languages (like C): use functions/procedures for abstraction
Data Abstraction	creating new data types that are well suited to an application by defining new classes
Encapsulation	<p>group data (attributes) and methods that manipulate the data to a single entity though defining a class</p> <ul style="list-style-type: none"> - class - packages
Information Hiding	<p><u>INFORMATION HIDING</u></p> <ul style="list-style-type: none"> - refers to hiding attributes and methods from the user of the class, e.g. using the private keyword.
Delegation	<p><u>DELEGATION THROUGH ASSOCIATION</u></p> <ul style="list-style-type: none"> - the process of assigning different responsibilities to different classes - 'has a' relationship -
Inheritance	<ul style="list-style-type: none"> - A form of abstraction that permits "generalisation" of similar attributes/methods of classes; analogous to passing genetics on to your children. - 'is a' relationship - classes and subclasses
Polymorphism	<p>Keyword</p> <p><i>Polymorphism:</i> The ability to use objects or methods in many different ways; roughly means "multiple forms".</p> <p>Overloading same method with various forms depending on signature (Ad Hoc polymorphism)</p> <p>Overriding same method with various forms depending on class (Subtype polymorphism)</p> <p>Substitution using subclasses in place of superclasses (Subtype polymorphism)</p> <p>Generics defining parametrised methods/classes (Parametric polymorphism, <i>coming soon</i>)</p>

2 CLASSES AND OBJECTS

- fundamental unit of abstraction in OOP; represents an 'entity'
- derived data type
- level of encapsulation

// Circle.java public class Circle { ... }	// Class Definition
public static int numCircles = 0;	<p>// <u>Static Variable</u></p> <ul style="list-style-type: none"> - shared between objects (1 copy per class)
private double radius;	<p>// <u>Instance Variable</u></p> <ul style="list-style-type: none"> - property/attribute unique to each instance/object (1 copy per object) - should be private

<pre>public Circle() { radius = 5.0; numCircles++ } public Circle(double radius) { setRadius(radius); numCircles++; }</pre>	<pre>// <u>Constructor</u> - method used to create and initialise an object - same name as the class - cannot return values // <u>Method Overloading</u> - defining methods with the same name but with different signatures (argument types and/or numbers)</pre>
<pre>public double getRadius() { return radius; }</pre>	<pre>// <u>Getter/Accessor</u></pre>
<pre>public void setRadius(radius) { this.radius = radius; }</pre>	<pre>// <u>Setter/Mutator</u> - this: refers to the calling object, the object that owns/is executing the method</pre>
<pre>public double computeArea() { double area = Math.PI * radius * radius; return area; }</pre>	<pre>// Method that returns a value // <u>Local Variable</u> - declared inside a method</pre>
<pre>public static void printNumCircles() { System.out.println("numCircles: " + numCircles); }</pre>	<pre>// <u>Static Method</u> - can only call other static methods - can only access static data - cannot refer to Java keywords (this, super, ...)</pre>

OBJECTS	
<ul style="list-style-type: none"> - instance of a class - contains state (values given to attributes) and dynamic information 	
<pre>// in CircleTest.java's main method Circle circleA; circleA = new Circle(); Circle circleB = new Circle();</pre>	<pre>// Declaring object (creates null reference) // Instantiating object - new: Directs the JVM to allocate memory for an object // declaring and instantiating in 1 line</pre>
<pre>circleA.radius = 2.0;</pre>	accessing instance variable
<pre>double area = circleA.computeArea();</pre>	invoking instance methods

REFERENCES AND GARBAGE COLLECTION	
Garbage Collection in Java	<p>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.</p> <p>Example:</p>

	<div style="display: flex; justify-content: space-around; align-items: flex-start;"> <div style="text-align: center;"> <p>Before Assignment</p> </div> <div style="text-align: center;"> <p>Before Assignment</p> </div> </div> <p>Q does not have a valid reference and, therefore, cannot be used in the future</p>

STANDARD METHODS	
- all located inside the object's file (Circle.java)	
<pre>public boolean equals(Circle circle) { return <i>boolean expression</i>; }</pre>	used when comparing two objects using the == operator - without equals(): == checks if <i>references</i> are equal
<pre>public String toString() { return <i>String expression</i>; }</pre>	used when printing object
<pre>public Circle(Circle <i>circle</i>) { if (<i>circle</i> == null) { System.out.println("Error: not a valid circle"); System.exit(0); } <i>this.radius</i> = <i>circle.radius</i>; }</pre>	copy constructor - decides what to do when: <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <pre>Circle c1 = new Circle(5.0); Circle c2 = new Circle(c1); // copy c1</pre> </div> NOTE: Circle c3 = c1 ~ makes c3 a reference to c1's object (doesn't use copy constructor)

JAVA PACKAGES	
- group of classes and interfaces	
Why would you group classes into packages?	- reusable - prevent naming conflicts - access control - another level of encapsulation
<pre>package <i>directoryname1.directoryname2</i> public class Circle { ... }</pre>	creating java package - eg. file structure: <i>parent/utilities/shapes</i> /Circle.java - parent directory of utilities must be in the CLASSPATH environment variable
import packageName.*	importing all classes in package
import packageName.className	importing particular class in package

INFORMATION HIDING	
- ability to hide the details of a class from the outside world - eg. TestCircle.java should not be able to see/use everything in Circle.java	
Access Control	Preventing an outside class from <u>manipulating</u> the properties of another class in <u>undesired ways</u>

VISIBILITY MODIFIER

Modifier	Class	Package	Subclass	Outside
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

- don't over-expose your classes (don't use public unless it's necessary)

public	- available/visible everywhere (within/outside the class)
protected	- visible/available to: <ul style="list-style-type: none"> - within the class - subclasses - all classes that are in the same package as the class - subclasses in other packages
default	
private	- only visible to methods <u>within the class</u>
Complete the table:	
Why should we use private for attributes in a class?	- to protect the value
What is a mutable class?	- a class that contains public mutator methods or
What is an immutable class?	- a class that contains no methods (other than constructors) that change any of the instance variables - does not contain set methods
Does mutability also involve static setters/getters?	No

DELEGATION THROUGH ASSOCIATION

- a class can delegate its responsibilities to other classes
- an object can invoke methods in other objects through containership
- association relationship between classes

Delegation - Example

```
public class Circle {
    private Point centre;
    private double radius;

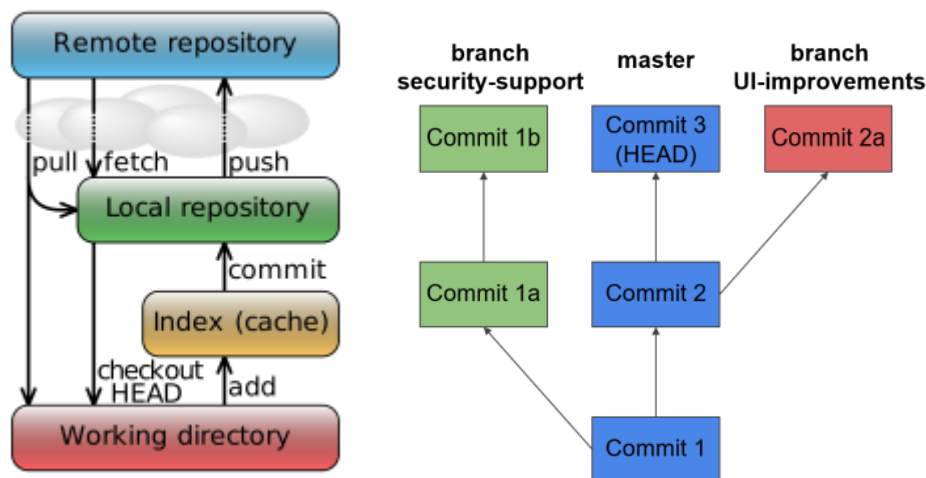
    public Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }
    public double getX() {
        return centre.getXCoord();
    }
    public double getY() {
        return centre.getYCoord();
    }
    // Other methods go here
}
```

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.

SOFTWARE TOOLS AND BAGEL (NOT EXAMINABLE)

GIT





- software version control



<code>git clone cloneURL</code>	creates a local copy of a remote repository
<code>git add .</code> <code>git add fileName</code>	stages all files in the directory stage a specific file
<code>git commit -m "message"</code>	adds files to local repository (working directory → staging cache → local repository)
<code>git push -u origin master</code>	add the changes to the master branch of the remote repository
<code>git status</code>	shows current status of your commit <ul style="list-style-type: none"> - files modified - which modifications are staged

git log	shows history of the repository <ul style="list-style-type: none"> - previous commits - branches
git branch	shows the current branch you are in
git reset -hard	goes back to previous commit version
<p style="text-align: center;">MAVEN</p> <ul style="list-style-type: none"> - manages software builds - automatically importing the required libraries and dependencies from external repositories - project structure declared in pom.xml <ul style="list-style-type: none"> - POM: Project Object Model 	
<p style="text-align: center;">INTELLIJ DEBUGGER</p> <ul style="list-style-type: none"> - find program bugs 	
<p style="text-align: center;">BAGEL</p> <ul style="list-style-type: none"> - Basic Academic Graphical Engine Library - game development - built by previous tutor of this subject - Documentation 	

6 INHERITANCE AND POLYMORPHISM	
Inheritance	A form of abstraction that permits “generalisation” of similar attributes/methods of classes. <ul style="list-style-type: none"> - defines an “is a” relationship <ul style="list-style-type: none"> - Rook is a Piece - Dog is an Animal - Husky is a Dog
SUPERCLASSES AND SUBCLASSES	
Superclass	Parent/base class - provide general information to its child classes
Subclass	Child/derived class - inherits common attributes/methods from parent class <ul style="list-style-type: none"> - contains all public/protected instance variables
<pre>public class SuperClass { ... }</pre> <div style="border: 1px solid black; padding: 10px; margin-top: 10px;"> <pre>public class Piece { private int currentRow; private int currentCol; // Constructor public Piece(int currentRow, int currentCol) { this.currentRow = currentRow; this.currentCol = currentCol; } }</pre> </div>	Defining superclass

<pre> public int getCurrentRow() {...} public void setCurrentRow(int currentRow) {...} } </pre>	
<pre> public class Subclass extends Superclass { ... } public class Rook extends Piece { // Constructor public Rook(int currentRow, int currentCol) { super(currentRow, currentCol); } // additional methods not in parent class public void move(...) {...} public boolean isValidMove(...) {...} } </pre>	<p>Creating subclasses</p> <ul style="list-style-type: none"> - extends: indicates one class inherits from another - super(...): invokes a constructor in the parent class <ul style="list-style-type: none"> - may only be used in subclass' constructor - must be the first statement in subclass constructor - parameter types must match
<h3>UPCASTING AND DOWNCASTING</h3>	
<p>Upcasting</p>	<p>object of a child class assigned to a variable of an ancestor class</p> <ul style="list-style-type: none"> - advantage: calls the generic method <pre>Piece p = new Rook();</pre>
<p>Downcasting</p>	<p>object of ancestor classes assigned to a variable of child class</p> <ul style="list-style-type: none"> - only makes sense if underlying object is actually of that class <pre>Piece robot = new WingedRobot(); WingedRobot plane = (WingedRobot) robot;</pre> <ul style="list-style-type: none"> - to access specific methods/attributes specific to the subclass
<pre> Subclass var0 = new Subclass(); Superclass var1 = new Subclass(); Subclass var2 = new Superclass(); Superclass var3 = new Superclass(); ReferenceType varName = new ObjectType(); </pre>	<p>Declaration and Initialisation</p> <pre> //  //  //  //  </pre> <p>when calling methods, we call methods overridden in</p>

	ObjectType def								
Why do we want Superclass var = new Subclass();?	<p>to store all Superclasses in a single array</p> <p>Eg.</p> <pre>Pieces rook = new Rook(); Pieces pawn = new Pawn(); Pieces[10] arr = ...</pre>								
OVERRIDING AND OVERLOADING									
<pre>// method in superclass public boolean isValidMove(...) {...} // method in subclass @Override public boolean isValidMove(...) {...}</pre>	<ul style="list-style-type: none"> - if method is only defined in parent class (no overriding) <ul style="list-style-type: none"> - method in parent class is called - regardless of object type - if method is overridden in child class <ul style="list-style-type: none"> - method called depends on type of object <table border="1"> <tr> <td></td><td>calls</td></tr> <tr> <td>Rook rook1 = new Rook(...);</td><td>subclass</td></tr> <tr> <td>Piece rook2 = new Rook(...);</td><td>subclass</td></tr> <tr> <td>Piece rook3 = new Piece(...);</td><td>superclass</td></tr> </table> <ul style="list-style-type: none"> - method called is based on type of <u>object</u>, not reference - <div style="border: 1px solid black; padding: 5px; display: inline-block;">Reference ... = new <u>Object</u>()</div> <p>@Override annotation</p> <ul style="list-style-type: none"> - optional - indicates that the method is overriding a methods in the parent class 		calls	Rook rook1 = new Rook(...);	subclass	Piece rook2 = new Rook(...);	subclass	Piece rook3 = new Piece(...);	superclass
	calls								
Rook rook1 = new Rook(...);	subclass								
Piece rook2 = new Rook(...);	subclass								
Piece rook3 = new Piece(...);	superclass								
<pre>// method in superclass public boolean isValidMove(...) {...} // method in subclass @Override public boolean isValidMove(...) { boolean isValid = true; // call the superclass first to do general logic if (!super.isValidMove(...)) { return false; } // add additional logic specific to this subclass if (...) { ... } return isValid; }</pre>	<ul style="list-style-type: none"> - super.... reference to an object's parent class; refers to attributes and methods of the parent 								

Overriding	Overloading
Declaring a method that exists in a superclass again in a subclass, with the same signature . Methods can only be overridden by subclasses .	Declaring multiple methods with the same name, but differing method signatures . Superclass methods can be overloaded in subclasses.
// method in superclass public boolean isValidMove(...) {...}	// in a class public Circle() { radius = 5.0; numCircles++ }
// method in subclass @Override public boolean isValidMove(...) {...}	public Circle(double radius) { setRadius(radius); numCircles++; }
Runtime Polymorphism	Compile time Polymorphism
cannot change argument types or number of arguments	can change number of arguments
cannot change return type	

RESTRICTING INHERITANCE

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

- private superclass functions cannot be @Overridden by functions in the subclass
 - `private` can only be accessed by the class
- [VISIBILITY MODIFIER](#)

<p>Keyword</p> <p><i>final</i>: Final methods may not be overridden by subclasses.</p> <pre> 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 }</pre> <p>This will restrict the move() method being overridden.</p>	<p>Final method</p> <ul style="list-style-type: none"> - cannot be overridden by subclasses - BUT child class can still access a `public final` method
<p>Why we should NOT create `protected` variables/methods</p>	<p>Privacy Leaks</p> <p>Defining attributes as protected allows updating them directly from classes.</p> <p>However, this should be avoided because it results in privacy leaks. attributes of the parent class should be accessed via public or protected methods in the parent class.</p> <p>Example:</p> <ul style="list-style-type: none"> • A good design of the Piece class should ensure that any method updates the attributes, currentRow, currentColumn, checks if 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 state of the object.
<p>Changing visibility modifiers when overriding</p>	<p>When overriding a method, a child class cannot further restrict the visibility of an overridden method. When overriding:</p> <ul style="list-style-type: none"> • 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 <p>public → protected → private</p>
<p>Shadowing</p>	<p>Keyword</p> <p><i>Shadowing</i>: 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.</p> <p>Don't. Do. It.</p> <p>You only need to define (common) variables in the superclass.</p>
<p>CHECKING OBJECT CLASSES</p>	
<p>getClass(obj)</p>	<p>returns the classname of</p>
<p>obj instance of class</p>	
<p>ABSTRACT CLASSES</p> <ul style="list-style-type: none"> - for parent classes that: <ul style="list-style-type: none"> - aren't actual entities - are abstract concepts - does not make sense to create an object of that type - can contain abstract (or non-abstract) methods - A class that 	

Concrete	Abstract
any class that is not abstract, and has well-defined, specific implementations for all actions it can take	represents common attributes and methods of its subclasses, but that is missing some information specific to its subclasses
Can be instantiated	Cannot be instantiated
Cannot have abstract methods	May have abstract methods
Represents entities that are part of the problem	Represents an incomplete concept

<pre>public abstract class Piece { // attributes, abstract methods... etc. }</pre>	creating abstract classes <ul style="list-style-type: none"> - prevents creation of object of type Piece - abstract: defines a class that is incomplete. abstract classes are general concepts, rather than being fully realised/detailed
--	--

ABSTRACT METHODS	
<pre>// abstract method public abstract boolean isValidMove(...);</pre>	Abstract Methods <ul style="list-style-type: none"> - abstract: defines a superclass method that is common to all subclasses but no implementation - each subclass then provides its own implementation through overriding - common across all subclasses but behaviour is different - no general method for all subclasses - specific to subclasses - subclasses must implement all abstract methods to be concrete

TYPES OF INHERITANCE	
Single Inheritance	only one superclass
Multiple Inheritance	(not in Java) several superclasses
Hierarchical Inheritance	one superclass, many subclasses

INTERFACE									
What is an interface?	<ul style="list-style-type: none"> - - 								
Interfaces VS Inheritance	<table> <tr> <th>Interfaces</th><th>Inheritance</th></tr> <tr> <td>"Can do" relationship</td><td>"Is a" relationship</td></tr> <tr> <td>usually named "...able"</td><td>Entity - usually named with noun</td></tr> <tr> <td></td><td></td></tr> </table>	Interfaces	Inheritance	"Can do" relationship	"Is a" relationship	usually named "...able"	Entity - usually named with noun		
Interfaces	Inheritance								
"Can do" relationship	"Is a" relationship								
usually named "...able"	Entity - usually named with noun								
<pre>public interface Printable { int MAXIMUM_PIXEL_DENSITY = 1000; void print(); }</pre>	<p>Defining interfaces</p> <ul style="list-style-type: none"> - methods have <u>no code</u> - all methods are <u>public</u> and <u>abstract</u> by default <ul style="list-style-type: none"> - using `default` makes it a 'normal' method - all attributes are <u>static final</u> - all methods and attributes are implied to be <u>public</u> 								
<pre>public class Image implements Printable { public void print() { <block of code to execute> } }</pre>	<p>Implementing interfaces</p> <ul style="list-style-type: none"> - concrete class must implement all methods defined in interface - classes that don't implement all methods, must be abstract 								
<p>Default Methods</p> <p>Classes can be "forced" to have an implementation of a method, that can then be overridden.</p> <pre>public interface Printable { default void print() { System.out.println(this.toString()); } }</pre> <p>Keyword</p> <p><u>default</u>: Indicates a <i>standard</i> implementation of a method, that can be overridden if the behaviour doesn't match what is expected of the implementing class.</p>									
	we can do inheritance and interfaces at the same time								
<pre>public interface Digitisable extends Printable { public void digitise(); }</pre> <ul style="list-style-type: none"> • Interfaces can be extended just like classes • Forms the same "Is a" relationship • Used to add additional, specific behaviour 	Extending interfaces								
SORTING (example)									
Arrays.sort(arrayOfThings)									
<p>public int compareTo(ClassName object)</p> <pre>public int compareTo(String string) { return this.length() - string.length(); }</pre>	Defines a method allowing us to order objects								

<p>Comparable Interface Example</p> <pre> 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); } } </pre>	
Comparable randomNumbers[] = new RandomNumber[10];	Upcasting with interfaces

8 MODELLING CLASSES AND RELATIONSHIPS https://creately.com/blog/diagrams/class-diagram-relationships/									
Pipeline	<ol style="list-style-type: none"> 1. Identify Classes <ul style="list-style-type: none"> ○ noun extraction 2. identify Class Relationships <ul style="list-style-type: none"> ○ identify relationships: <ol style="list-style-type: none"> i. has-a ii. is-a iii. can-do 3. Refine Classes and Relationships 4. Develop a Class Diagram <ul style="list-style-type: none"> ○ combine classes and relationships 5. Represent using an accepted notation <ul style="list-style-type: none"> ○ UML 								
UML									
UML	a graphical modelling language that can be used to represent object oriented analysis, design and implementation								
CLASS									
Visibility Modifiers / Access Control	<table border="1"> <tr> <td>+</td><td>public</td></tr> <tr> <td>~</td><td>default (package-private)</td></tr> <tr> <td>#</td><td>protected</td></tr> <tr> <td>-</td><td>private</td></tr> </table>	+	public	~	default (package-private)	#	protected	-	private
+	public								
~	default (package-private)								
#	protected								
-	private								

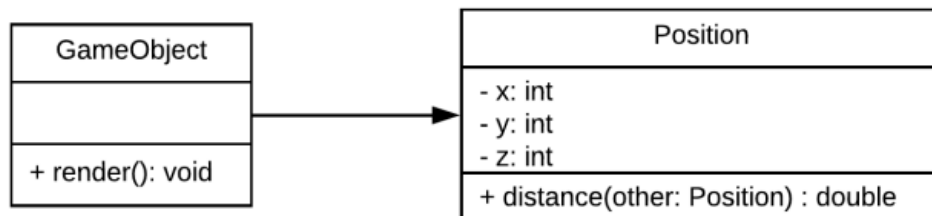
RELATIONSHIPS

Relationships in UML

Type	relationship
Association	has-a
Generalisation	inheritance
Realisation	
Dependency	

- Association
- Generalisation
- Realisation
- Dependency

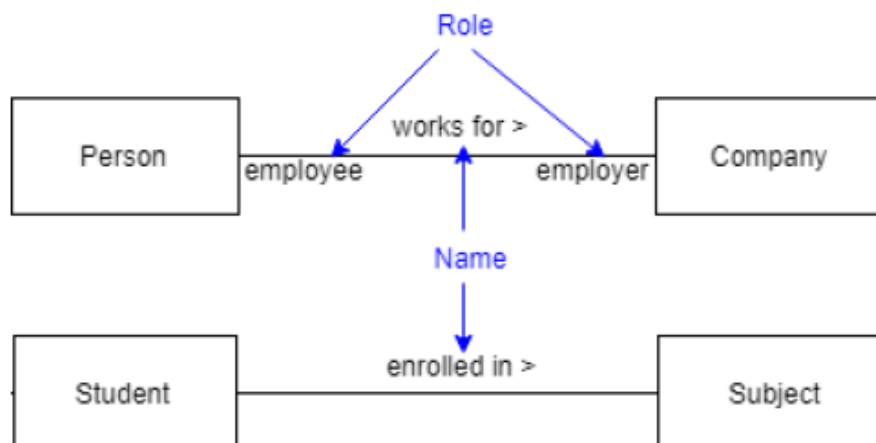
ASSOCIATION

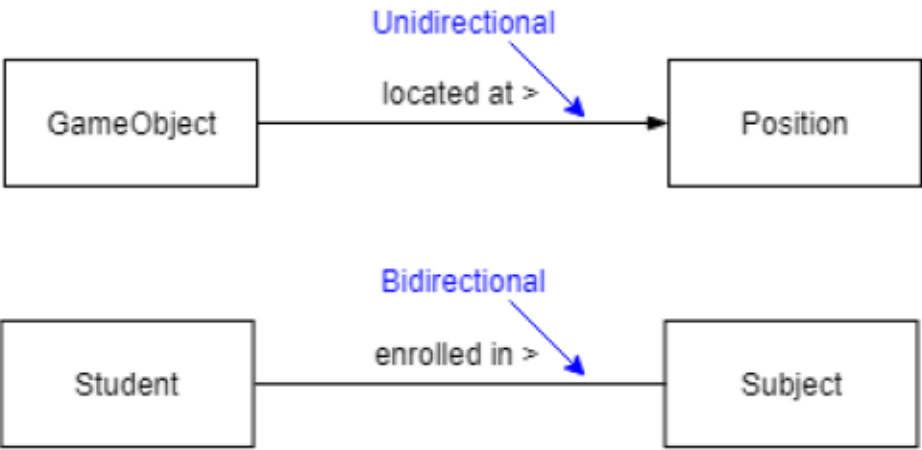
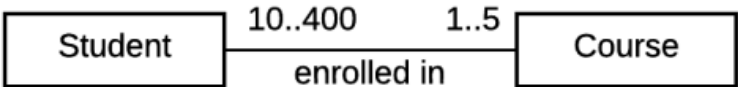
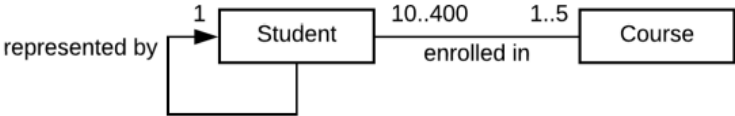


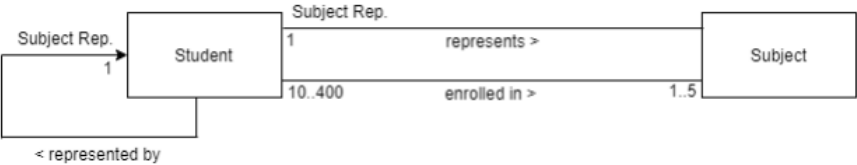
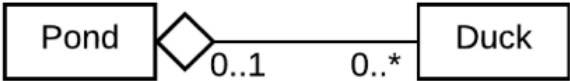
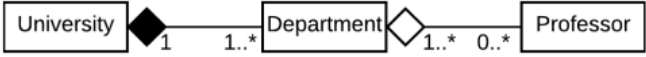
Association

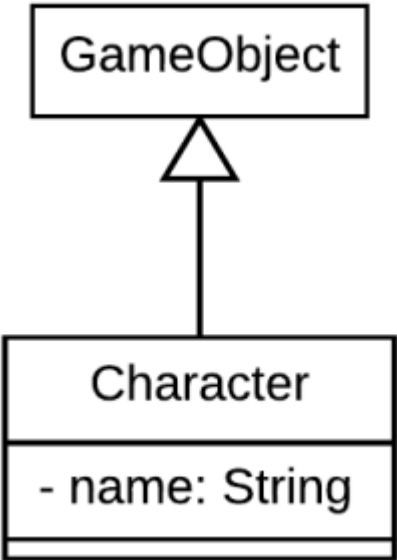
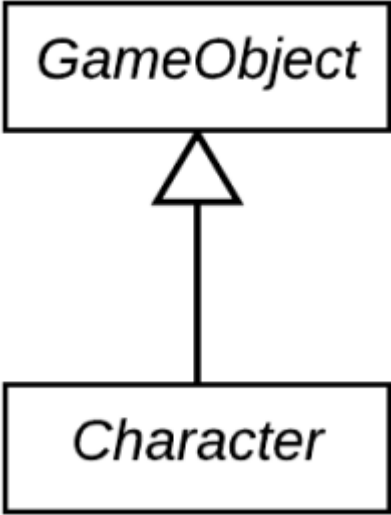
A link indicating one class contains an attribute that is itself a class. Does not mean that class “uses” another(in a method, or otherwise)

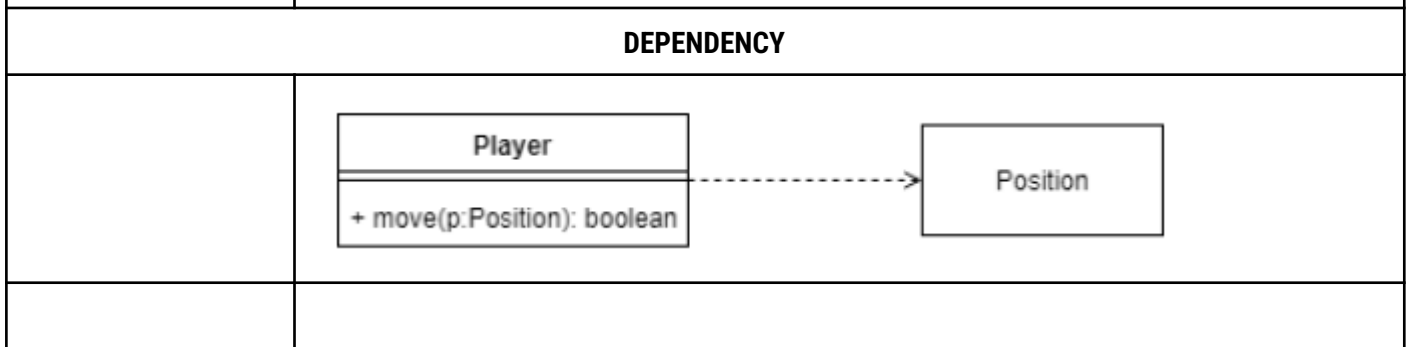
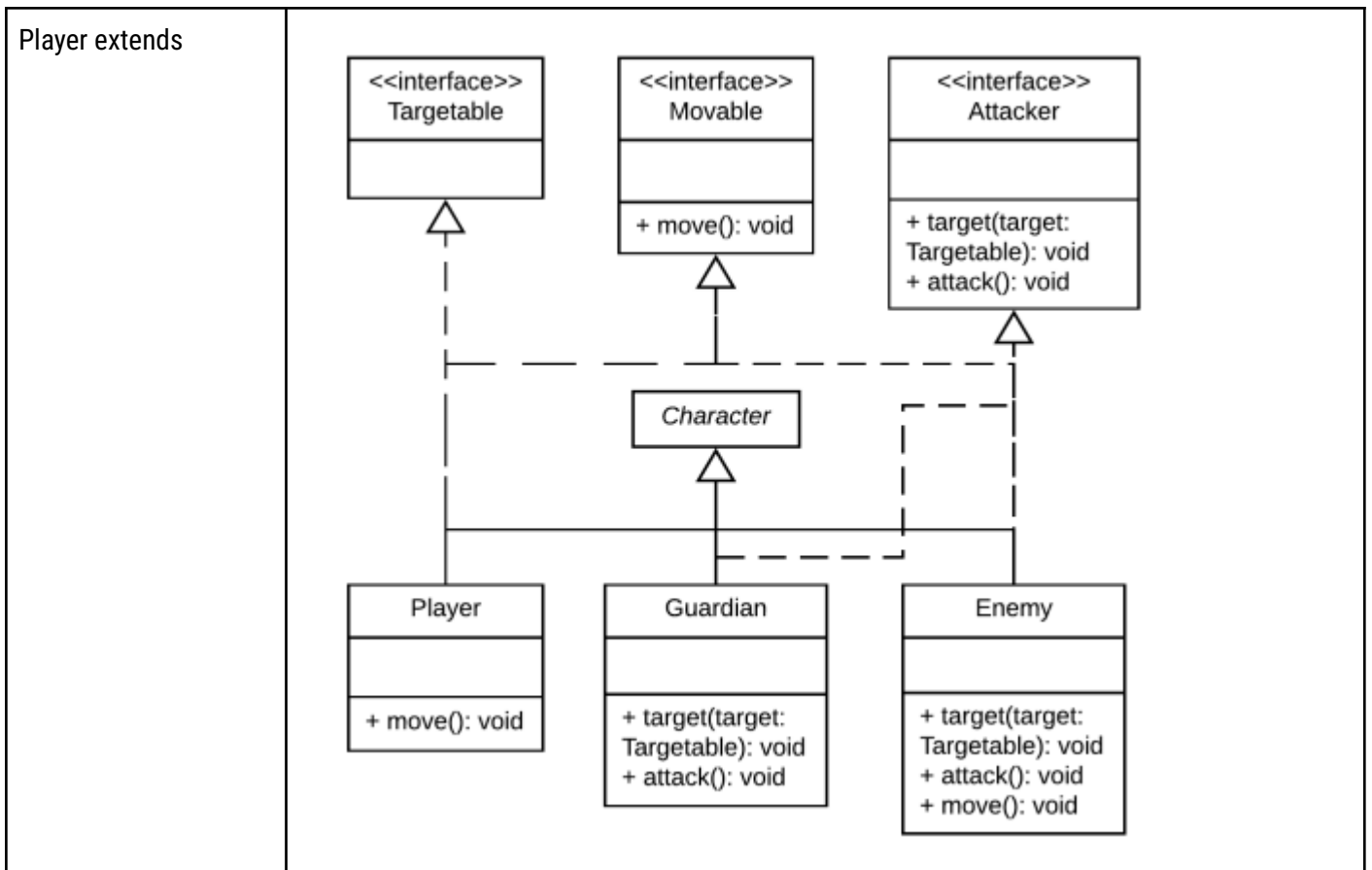
Name and Role



Direction	 <p>Unidirectional</p> <p>GameObject located at Position</p> <p>Bidirectional</p> <p>Student enrolled in Subject</p>								
Multiplicity	<p>specified the number of linked that can exist between instances (objects) of the associated classes</p> <ul style="list-style-type: none"> how many objects of one class relate to one object of another class <table border="1" data-bbox="400 777 1353 1039"> <tr> <td>Exactly one</td><td>1</td></tr> <tr> <td></td><td>0..1</td></tr> <tr> <td></td><td>0..* or *</td></tr> <tr> <td></td><td></td></tr> </table> <p>Create a UML representation for the following scenario:</p> <ul style="list-style-type: none"> 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 	Exactly one	1		0..1		0..* or *		
Exactly one	1								
	0..1								
	0..* or *								
Self Association	<p>Each Student has a student representative they can contact, who is also a student</p> 								

<p>Multiple Association</p>	<p>Does the following class relationship enforce a single student rep per-subject?</p> <p>How about a single student rep per-subject, per-student?</p> 
<p>Type (Aggregation)</p>	<p>Different form of association, where one class “has” another class, but both exist independently</p>  <ul style="list-style-type: none"> • 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
<p>Type (Composition)</p>	<p>Different form of association, indicating one class cannot exist without the other; in other words, existing on its own doesn't make sense</p>  <ul style="list-style-type: none"> • 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!
<p>GENERALISATION</p>	

	 <pre>classDiagram class GameObject class Character { - name: String } Character -- > GameObject</pre>
Abstract Classes	 <pre>classDiagram class GameObject class Character Character -- > GameObject</pre> <ul style="list-style-type: none">• Italicised
<div>• Interfaces</div> <div>• uses dashed lines</div> <div>REALISATION</div>	



9 GENERICS	
	<ul style="list-style-type: none"> - parameter types - a kind of polymorphism
What do generics enable us to do?	<ul style="list-style-type: none"> - allows generic logic to be written that applies to any class type - allows code reuse
What is `T`?	<ul style="list-style-type: none"> - a type parameter/type variable/parameterized type - any class or interface (Integer, String, Robot, Comparable, etc.) <ul style="list-style-type: none"> - <i>cannot</i> be a primitive type (int, float, etc.)
If we have a generic class, method or interface, must the types always be specified?	Yes
Why do we need generics?	<ul style="list-style-type: none"> - to prevent bad software design - Without generics:

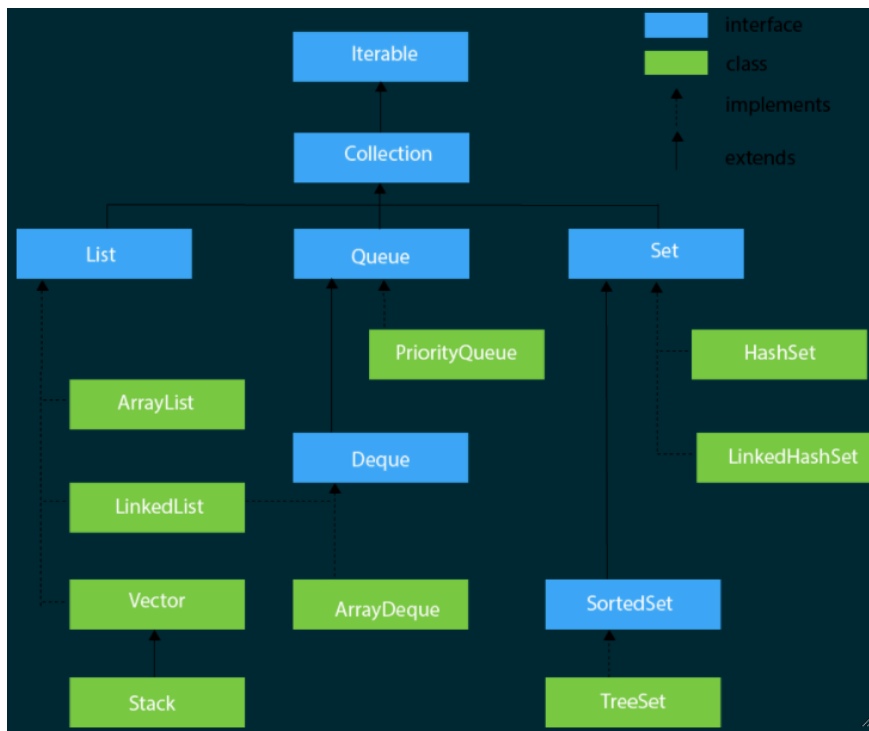
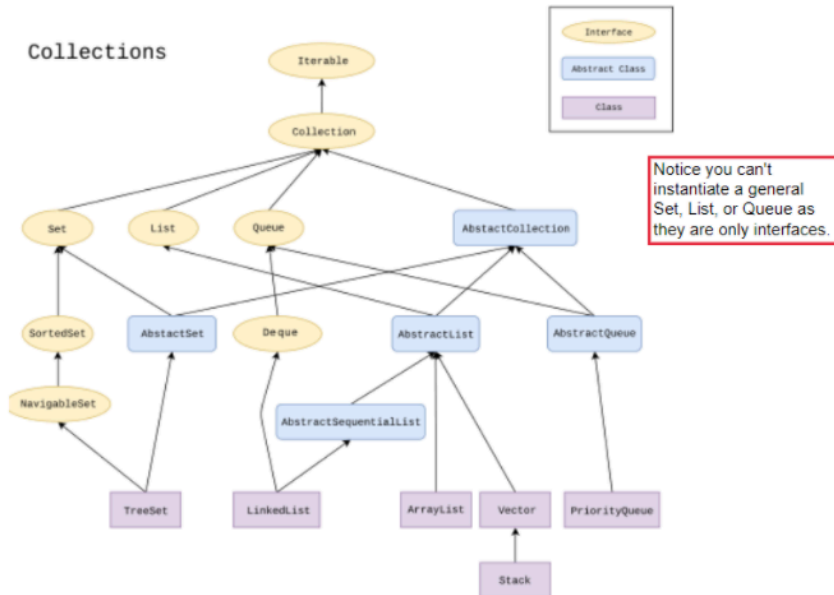
	<pre> 1 public class Circle implements Comparable { 2 private double centreX = 0.0, centreY = 0.0; 3 private double radius = 0.0; 4 @Override 5 public int compareTo(Object o) { 6 Circle c = null; 7 if (o instanceof Circle) { 8 c = (Circle)o; 9 if (c.radius > this.radius) 10 return 1; 11 else if (c.radius < this.radius) 12 return -1; 13 else 14 return 0; 15 } else { 16 return -2; 17 } 18 } 19 } </pre> <pre> 1 public class Square implements Comparable{ 2 private double centreX = 0.0, centreY = 0.0; 3 private double length = 0.0; 4 @Override 5 public int compareTo(Object o) { 6 Square s = null; 7 if (o instanceof Square) { 8 s = (Square)o; 9 if (s.length > this.length) 10 return 1; 11 else if (s.length < s.length) 12 return -1; 13 else 14 return 0; 15 } else { 16 return -2; 17 } 18 } </pre> <ul style="list-style-type: none"> - two compareTo functions for different comparable functions with similar if conditions - With generics: <pre> 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; } } </pre>
Limitations of generics	<ul style="list-style-type: none"> • instantiate parameterised objects <ul style="list-style-type: none"> ○ T item = new T(); • create arrays of parametrised objects <ul style="list-style-type: none"> ○ T[] elements = new T[];
GENERIC CLASS	
Single Type	<pre> public class SampleClass<T> { private T data; public void setData(T data) { this.data = data; } public T getData() { </pre>

	<pre> return data; } }</pre>
Multiple Types	<pre> public class SampleClass<T1, T2> { private T1 first; private T2 second public Sample(T1 first, T2 second) { this.first = first; this.second = second; } }</pre>
Bounded Type Parameters	<pre> # T must implement the Comparable Interface public class SampleClass<T extends Comparable<T>> {...} # T must be a subclass of Robot public class SampleClass<T extends Robot> {...} # T must extend multiple things public class SampleClass<T extends Robot & Comparable<T> & List<T>></pre>
<p style="text-align: center;">GENERIC METHOD</p> <p>- method that accepts argument, or return objects, of an arbitrary type</p>	
Defining Generic Methods	<pre> // Generic Argument public <T> int SampleMethod(T arg); // Generic return value public <T> T SampleMethod(String name); // Both public <T, S> T SampleMethod(S arg);</pre>
Does a generic method containing class have to be generic?	No

10 COLLECTIONS AND MAPS

COLLECTIONS

Collections



coll.size()	length of collection (c)
coll.contains(ele)	returns <i>true</i> if collection contains element and <i>false</i> otherwise
ARRAYLIST	
.add(index, ele) .add(ele) addAll(index, coll) .addAll(coll)	
.set(index, ele)	
.get(index)	
.indexOf(ele) .lastIndexOf(ele)	
.remove(index)	

.remove(ele)	
.subList(fromIndex, toIndex)	fromIndex (inclusive) - toIndex (not included)

<p>•</p> <h2>11 DESIGN PATTERNS</h2>	

<h2>12 EXCEPTIONS</h2>	
<h3>ERRORS</h3>	
Syntax Errors	<ul style="list-style-type: none"> - illegal code (causes compilation error)
Semantic Errors	<ul style="list-style-type: none"> - code runs but output is wrong - kind of like logic error - but not the same
Runtime Errors	<ul style="list-style-type: none"> - causes program to end prematurely - identified through execution - examples <ul style="list-style-type: none"> - dividing by zero - out of bounds of array - store incompatible data elements - negative value for array size - convert string to another type - file errors <ul style="list-style-type: none"> - opening file that does not exist / no read permission - writing to file without write permission
Compilation Errors	

<h3>ERROR HANDLING</h3>	
Defensive Programming	<p>using guard if statements</p> <pre> 1 public int divide(int n1, double n2) { 2 if (n2 != 0) { 3 return n1/n2; 4 } else { 5 ??? 6 } 7 } </pre> <pre> 1 if (n2 != 0) { 2 divide(n1, n2); 3 } else { 4 // Print error message and exit or continue 5 } </pre> <p>Solution 2: Explicitly guard yourself against dangerous or invalid conditions, known as <i>defensive programming</i>.</p>

Disadvantages of defensive programming	<ul style="list-style-type: none"> - need to explicitly protect against every possible error condition - some conditions don't have a "backup" or alternative path - they're just failures - not very nice to read (many if-else-if-else statements) - poor abstraction (bloated code)
Try-catch-finally block	<pre> 1 public void method(...) { 2 try { 3 <block of code to execute, 4 which may cause an exception> 5 } catch (<ExceptionClass> varName) { 6 <block of code to execute to recover from exception, 7 or end the program> 8 } finally { 9 <block of code that executes whether an exception 10 happened or not> 11 } 12 } </pre> <div style="border: 1px solid red; padding: 2px; width: fit-content; margin-top: 10px;">The finally block is optional.</div> <ul style="list-style-type: none"> - try <ul style="list-style-type: none"> - block of code to execute, which may cause an exception - catch <ul style="list-style-type: none"> - block of code to execute to recover from exception or end the program - finally (opt.) <ul style="list-style-type: none"> - run regardless of whether an exception happened or not - clean up (eg.
Exceptions	<ul style="list-style-type: none"> - Exception <ul style="list-style-type: none"> - an error state created by a runtime error in code - a Java Object: represents the error that has encountered <pre> } catch(Exception e) { System.out.println(e.getMessage()); } </pre> <ul style="list-style-type: none"> - (try not to catch Exception - catch more specific Exceptions) - Exception Handling <ul style="list-style-type: none"> - Code that actively protects our program in the case of exceptions
Chaining Exceptions	<pre> 1 public void processFile(String filename) { 2 try { 3 ... 4 } catch (FileNotFoundException e) { 5 e.printStackTrace(); 6 } catch (IOException e) { 7 e.printStackTrace(); 8 } 9 } </pre> <p>We can also <i>chain</i> catch blocks to deal with different exceptions <i>separately</i>. The most "specific" exception (subclasses) come first, with "broader" exceptions (superclasses) listed lower.</p> <ul style="list-style-type: none"> - put the most specific one first <ul style="list-style-type: none"> - [parent] IOException <ul style="list-style-type: none"> - [child] FileNotFoundException - because first catch is run first <ul style="list-style-type: none"> - ie. if IOException is before FileNotFoundException: the FileNotFoundException will never be run

GENERATING EXCEPTIONS

throw VS throws

throw	throws
Respond to an error state by creating an exception object, either already existing or one defined by you.	Indicates a method/class has the potential to create an exception, and can't be bothered to deal with it, or that the exact response varies by application.

throw

```
public Person(int age, String name) {
    if (name == null) {
        throw new NullPointerException("Creating person with null name");
    }
}
```

- handling the custom throw

```
try {
    Person p1 = new Person(10, "Sarah");
    System.out.println("Created object p1");
    Person p2 = new Person(12, null);
    System.out.println("Created object p2");
} catch (NullPointerException e) {
    System.out.println("Failed to create object");
}
```

throws

```
public class Circle {
    private double centreX, centreY;
    private double radius;

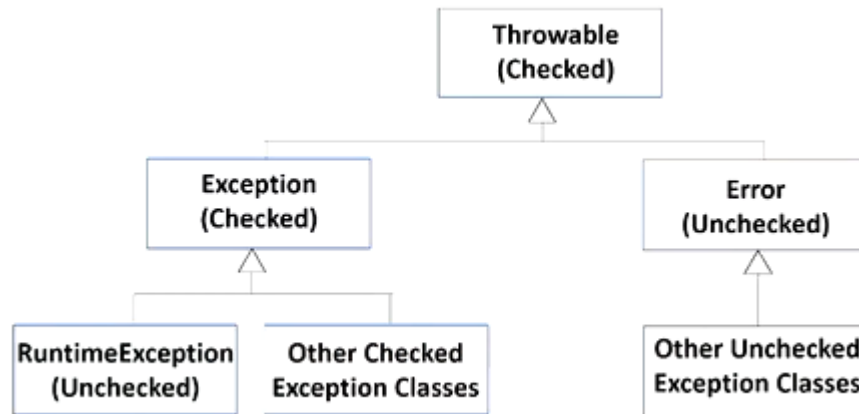
    public Circle (double centreX, double centreY, double radius)
        throws InvalidRadiusException {
        if (r <= 0 ) {
            throw new InvalidRadiusException(radius);
        }
    }
}
```

defining exceptions

- extending Exception class to create Exceptions
 - for your unique code and unique errors
- NOTE: 2 constructors for exception

```
1 import java.lang.Exception;
2
3 public class InvalidRadiusException extends Exception {
4     public InvalidRadiusException() {
5         super("Radius is not valid");
6     }
7
8     public InvalidRadiusException(double radius){
9         super("Radius [" + radius + "] is not valid");
10    }
11 }
```

TYPES OF EXCEPTIONS



Unchecked

- most exceptions are unchecked
- **can be safely ignored** by programmer

Checked

- must be explicitly handled by the programmer in some way
- compiler **gives error** if ignored

<https://howtodoinjava.com/java/exception-handling/checked-vs-unchecked-exceptions-in-java/#3-checked-exception-vs-unchecked-exception>

13 SOFTWARE TESTING AND DESIGN

14 EVENT DRIVEN PROGRAMMING

What is a Paradigm?

SEQUENTIAL PROGRAMMING

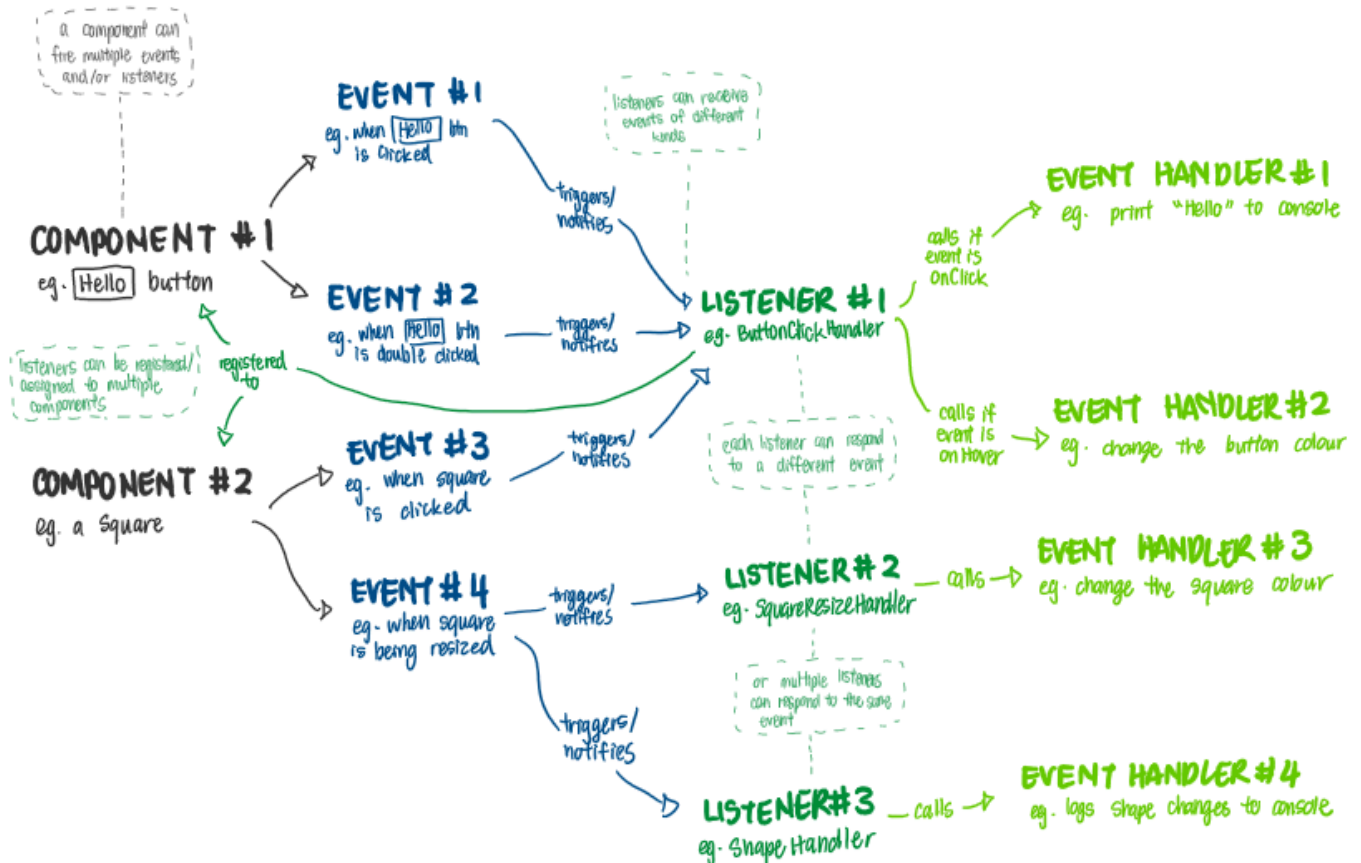
Sequential Programming

- one example of a programming paradigm
- A *static* program that is run (more or less) top to bottom, starting at the beginning of the main method, and concluding at its end
-

Static (in this context)

- Constant, unchangeable logic
- Execution is the same (or very similar) each time

EVENT-DRIVEN PROGRAMMING



Event-Driven Programming

- Using *events* and *callbacks* to control the flow of a program's execution based on changes to the program state
- signal-and-response approach
- asynchronous
 - programmer deals with events that are generated at unknown times

Events	<ul style="list-style-type: none"> - things that happen (signal) - created when state of object/device/etc. is altered - <i>firing</i> event: sending of event
Callbacks / Listeners (Java)	<ul style="list-style-type: none"> - methods/behaviour to execute when an event happens (reponse) - triggered by event
State	<ul style="list-style-type: none"> • the properties that define an object or device; for example, whether it is "active"

- component can have multiple listeners
- each listener may respond to a different kind of event, or multiple listeners may respond to the same event
- listener must register with event generator in advance

Events, Listeners and Event Handlers

Examples

- Exception handling
 - eg. exception (event) handled using xxx behaviour (callback)
- Observer pattern
 - eg. observing a button: constantly checking if button is pressed - if yes,

	<p>carry out xxx behaviour (callback)</p> <ul style="list-style-type: none"> Graphical User Interfaces (GUI) <ul style="list-style-type: none"> similar to above eg. in Java GUI Framework JavaFX, Web Development Frameworks Embedded Systems/Hardware 				
SOFTWARE DEVELOPMENT FRAMEWORKS					
Software Development Frameworks	<ul style="list-style-type: none"> set of cooperating classes that represent reusable designs consists of <ul style="list-style-type: none"> abstract classes (partially complete) and interfaces <ul style="list-style-type: none"> that you will implement partially complete classes that can be customised to meet application needs 				
Key Features	<ul style="list-style-type: none"> Extensibility Inversion of Control <ul style="list-style-type: none"> <table border="1"> <thead> <tr> <th>Conventional Library</th><th>Application Development Framework</th></tr> </thead> <tbody> <tr> <td>application controls the flow of execution – application acts as the master</td><td> flow of execution resides in the framework – framework acts as the master <ul style="list-style-type: none"> not as much control over what we can do with the button we made </td></tr> </tbody> </table> Design Patterns as building blocks 	Conventional Library	Application Development Framework	application controls the flow of execution – application acts as the master	flow of execution resides in the framework – framework acts as the master <ul style="list-style-type: none"> not as much control over what we can do with the button we made
Conventional Library	Application Development Framework				
application controls the flow of execution – application acts as the master	flow of execution resides in the framework – framework acts as the master <ul style="list-style-type: none"> not as much control over what we can do with the button we made 				

15 ADVANCED JAVA CONCEPTS	
Enumerated Types	
<pre>public enum Colour { RED, BLACK }</pre>	<p>enum: A class that consists of a finite list of constants</p> <ul style="list-style-type: none"> represent fixed set of values list <i>all</i> values
Making Cards with suits that are tied to colour	<pre>public enum Suit { SPADES(Colour.BLACK) CLUBS(Colour.BLACK) DIAMOND(Colour.RED) HEART(Colour.RED) private Colour colour; private Suit(Colour colour) { this.colour = colour } }</pre> <ul style="list-style-type: none"> enum objects are treated just like any other object
<pre>Colour colour = Colour.RED</pre>	Enum Variables <ul style="list-style-type: none"> values accessed statically throws error if value is not in enum
Standard Methods	<ul style="list-style-type: none"> default constructor, toString(), compareTo(), ordinal() can be overridden (like any other class)
Variadic Parameters	
Variadic Method	<ul style="list-style-type: none"> method that takes in <i>unknown number</i> of arguments

	<ul style="list-style-type: none"> eg. Arrays.asList <div> <pre> List<Integer> list1 = Arrays.asList(12, 5); System.out.println(list1); List<Integer> list2 = Arrays.asList(12, 5, 45, 18); System.out.println(list2); List<Integer> list3 = Arrays.asList(12, 5, 45, 18, 33); System.out.println(list3); </pre> </div> <div>Different number of parameters each time for println() statements.</div>
Creating Variadic Parameter	<pre> public static String concatenate(Strings... strings) {...} public static double average(int... nums) {...} </pre> <div> <pre> 1 public class variadicExample2 { 2 public static void main (String[] args) { 3 System.out.println(concatenate("Hello", "world!")); 4 System.out.println(concatenate("Programming", , "is", "fun!")); 5 } 6 7 public static String concatenate(String... strings) { 8 String string = ""; 9 for (String s : strings) { 10 string += " " + s; 11 } 12 return string; 13 } 14 } </pre> </div> <p>Output:</p> <pre> Hello world! Programming is fun! </pre>
FUNCTIONAL INTERFACES AND LAMBDA EXPRESSIONS	
Functional Interfaces	<ul style="list-style-type: none"> An interface that contains only a single abstract, non-static method <div> <pre> @FunctionalInterface public interface Attackable { public void attack(); } </pre> </div> a tool used with other techniques Examples: Predicate, Unary Operators

Example: Predicate

```
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

```
eg.  
// Create predicate  
Predicate<Integer> lesserthan = (i) -> (i < 16);  
  
// Call Predicate method  
System.out.println(lesserthan.test(9))
```

Example 2: Unary Operator

```
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

minus	-
NOT	!
increment	++
decrement	--
bitwise complement	~

```
UnaryOperator<Integer> increment = (i) -> (i += 1);  
int x = 10;  
System.out.println(increment.apply(x));
```

LAMBDA EXPRESSIONS

Lambda Expressions

- short block of code which takes in parameters and returns a value
- instances of functional interfaces
 - allow us to treat the functionality of the interface as an object

```
// parameter -> expression  
(param1, param2, ...) -> { code block }
```

- can take in zero or more arguments
- recommended to use brackets
- there are restrictions for code block
 - no for loops
 - no creating variables
 - ...

Examples

- Doubling an integer
- Comparing two objects

	<ul style="list-style-type: none"> Performing a boolean test on an object Copying an object ...
Anonymous Classes VS Lambda Expressions	<ul style="list-style-type: none"> lambda is newer and simpler <p>Anonymous Class</p> <pre>starWarsMovies.sort(new Comparator<Movie> { public int compare(Movie m1, Movie m2) { return m1.rating - m2.rating; } });</pre> <p>Lambda Expression</p> <pre>starWarsMovies.sort((m1, m2) -> m1.rating - m2.rating);</pre>
METHOD REFERENCES	
Method References	<pre>names.replaceAll(String::toUpperCase);</pre> <p>Keyword</p> <p><i>Method Reference:</i> An object that stores a <i>method</i>; can take the place of a lambda expression if that lambda expression is only used to call a single method.</p> <p>Method references can be <i>stored</i> in the same way a lambda expression can:</p> <pre>UnaryOperator<String> operator = s -> s.toLowerCase();</pre> <p>lambda expression</p> <pre>UnaryOperator<String> operator = String::toLowerCase;</pre> <p>method reference</p>
Examples	
JAVA STREAMS	
	<ul style="list-style-type: none"> A series of elements given in sequence, that are automatically put through a pipeline of operations.
function VS streams	function

```

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());
        }
    }
    return newStrings;
}

```

VS
stream

```

list = list.stream()
    .filter(s -> s.length() > 5)
    .filter(s -> s.startsWith("C"))
    .map(String::toUpperCase)
    .collect(Collectors.toList());

```

Stream Operations

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