Object Oriented Programming
SWEN20003

What is Java?
- Both compiles and can be interpreted
- **Portable:** "Write once, run anywhere"
- Object Oriented

Applications
- General purpose stuff
- Java webapps
- Android app development

Hello World in Java:

[java]
```
// Print out a friendly greeting
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```
[/java]

!Class:
- Fundamental <building block> of all Java programs.
- One or more classes are used to model/represent data or items in a problem
- This example would be in Hello.java
- <Everything must be in a class>

!Method:
- Basic unit of abstraction in Java (after classes)
- Somewhat equivalent to functions
- <Java requires a main method in order for code to work>
- Need one main method defined in each set of classes

!Variables and Data Types:
- Variables store information for later use/manipulation
- Every variable has a type like in C

!Object:
- Classes are definitions, <whereas objects are real and hold data>

Java statements must end with semicolons;

Comments are done with //

!Privacy:
- A.K.A. Information Hiding or Access Control
- One of the most important aspects of software

!Static:

- Determines whether a variable/method <requires an object to be made> before accessing/using it.

Primitive Types:
- Building Blocks: All data are build from primitives
- Primitives can't be broken into smaller parts

Type	Bytes	Values
<boolean>	1	<false, true>
<char>	2	All unicode characters (e.g. 'a')
<byte>	1	-2^7 to 2^7-1 (-128 to 127)
<short>	2	-2^{15} to $2^{15}-1$ (-32768 to 32767)
<int>	4	-2^{31} to $2^{31}-1$
<long>	8	-2^{63} to $2^{63}-1$
<float>	4	$\approx \pm 3x10^{38}$ (limited precision)
<double>	8	$\approx \pm 10^{308}$ (limited precision)

Java convention is <camelCase>

Strings

A string is a <class> and a <datatype>
Every class in java can be made into a data type

Append strings to each other in print statements with "+"

[java]
```
System.out.println("1 + 1 = " + 1 + 1);
```
[/java]
Actually prints "1 + 1 = 11"
Use brackets to get around this

Some methods to know:
- .length();
- .toUpperCase();
- .toLowerCase();
- .substring(int, int);
- .replace(char, char);

Strings are immutable, so you'll need to assign them back to themselves if you use a method

<Never use == to compare objects!!!!>
It doesn't work because pointers

!Wrapper Class:
A class that gives extra functionality to primitives like <int> and lets them

behave like objects

This is where things like <Integer.parseInt()> come from.

Formatting in java:
[java]
System.out.format(Specifier, Arg Index, Flags, Width, Precision, Conversion);

System.out.format("%3.2f", 4.56789);
// Prints 4.57
[/java]

IO

<Input>
Java has a powerful approach to input called the Scanner

[java]
import java.util.Scanner;

//Create the scanner
Scanner scanner = new Scanner(System.in);
[/java]

<Only ever create one Scanner for each program or bad things happen>
"new" allocates memory for an object and creates it (just like malloc)

Some Scanner methods:
- .nextLine();
- .nextInt();
- .nextDouble();
- .nextBoolean();

[java]
// Example of using buffered reader and file reader
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class Program {

    public static void main(String[] args) {

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

            String text;

```
            while ((text = br.readLine()) != null) {
                <block of code to execute>
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
[/java]

Could also use scanner:
[java]
try (Scanner scanner = new Scanner(new FileReader("test.txt"))) {
    while (scanner.hasNextLine()) {

    }
}
[/java]


<Output>

[java]
try (PrintWriter pw = new PrintWriter(new FileWriter("test.txt")));
[/java]

The above creates two objects:
- FileWriter - a low level file for simple character output
- PrintWriter - a higher level file that allows for more sophisticated
formatting (same methods as System.out)

<try> will automaticall close the file once we're done
<pw> is our file variable

pw.print - Outputs a string
pw.println - Outputs a string with a new line
pw.format - Output a string and allows for format specifiers

catch - acts as a safeguard to potential errors - prints an error message

--------------------------------------------------------------------------------

                                Arrays

--------------------------------------------------------------------------------

Arrays are references
Manipulating one reference affects all references

Java permits "multi-dimensional" arrays
Technically exist as "array of arrays"

[java]
```

```
int[][] nums = new int[10][10];      // Square array
int[] n1 = {1, 2, 3};                // initialise an array
ints = new int[ints.length + 1];     // resizing
[/java]

Make sure to compare arrays you use .equals(n1, n2);

---

!Static:
Indicates a constant, variable, or method exists without an object. In other
words, you do not need to create a variable to use something defined as static.

[java]
public class Program {
    public static final int N_ELEMENTS = 10;

    public static void main(String args[]) {
        int elements[] = new int[N_ELEMENTS];
    }
}
[/java]

- N_ELEMENTS is a class constant, available to any method defined in the Program
    class
- Defined in the class but <outside a method>
- Somewhat equivalent to C's #define

[java]
public class Program {
    public static final int N_ELEMENTS = 10;

    public static int computeValue(int x) {
        ...
    }
}
[/java]

- computeValue is a class method, available to any other (static) method defined
in the Program class
- Defined <in the class>
- Can only call/use other static methods and variables

---

                          Classes and Objects

---

!Class:
- Fundamental unit of abstraction in Java. Represents an "entity" that is part
of a problem.

!Objects:
```

- An instance of class
- Contain state, or dynamic information

!Instance:
- An object that exisxts in your code

!Class (Static) Variable:
- A property or attribute that is <common to all instances of class>
- Can be a constant or variable
- One copy per class (e.g. count of a database)

!Instance Variable:
- A property or attribute that is unique to each instance of a class
- One copy per object

!Class (Static) Method:
- An action that can be performed by a class or a message that can be sent to it
- Defines an action performed by a class
- <Does not refer to any instance variables>

!Instance Method:
- An action that can be performed by an object, or a message that can be sent to it
- Defines an action performed by an object

If a method doesn't use any instance variables it should be static.

```
# Useful Rule of Thumb™: Make all methods static, and then remove static only if
# remove static only if the method uses an instance variable.
```

!Instantiating a Class:
- To create an object of a class
- Must use <new>

e.g.

```java
Animal dog = new Animal();
```

!Constructor:
- A method used to create and initialise an object

!this:
- A reference to the calling object, the object that owns/is executing the method

Standard Methods we should have
- equals
- toString
- getters and setters

Privacy and Mutability

!Mutable:
- An object is mutable if any of its instance variables can be changed after being initialised.

!Immutable:
- An object is immutable if none of its instance variables can be changed after being initialised

!Information Hiding:
- Using privacy to hide the details of a class from the outside world

!Access Control:
- Preventing an outside class from manipulating the properties of another class in undesired ways

!Private:
- Only available to methods defined in the same class; should be applied to almost all (mutable) instance variables and some methods.

!Package:
- Default, applied if nothing else specified; available to all classes in the same package

!Protected:
- Available to all classes in the same package and also to any subclasses that inherit from the package

!Public:
- Available at all times, always

Modifier	Class	Package	Subclass	World
<public>	Y	Y	Y	Y
<protected>	Y	Y	Y	N
<default>	Y	Y	N	N
<private>	Y	N	N	N

!Getters/Accessor/Setter/Mutator
- Basically give access to instance variable from a class

!Privacy Leak:
- When a reference to a private instance variable is made available to an external class, and uninstended/unknown changes can be made

When returning an object, make sure to return a copy!

!Immutable:
- A class is immutable if all of its attributes are private, it contains no setters, and only returns <copies> of its instance variables
(i.e. instance variables cannot be externally altered after creation)

```
--------------------------------------------------------------------------------

                                Inheritance

--------------------------------------------------------------------------------

!Inheritance:
- A form of abstraction that permits generalisation of similar attributes
methods of classes; analogous to passing genetics on to your children

!Subclass:
- The "child" or "derived" class in the inheritance relationship; inherits
common attributes and methods from the "parent" class

!Superclass:
- The "parent" or "base" class in the inheritance relationship; provides general
information to its "child" classes

Inheritance defines an "Is A" relationship
Only use inheritance when this relationship makes sense

!super:
Invokes a constructor in the parent class

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 must match that of the constructor in the
base class

!Shadowing:
- When two or more variables are declared with the same name in overlaping
scopes; for example in both a subclass and a superclass
# Don't do it.

!Overloading:
- When you declare multiple methods with the same name, but differing method
signatures
- Superclass methods can be overload in subclass

[java]
public class Cheese {
    // This is overloading
    public void print(String fish) {
        System.out.println(fish);
    }

    public void print(String fish, int rainbow) {
        // but what if you want to print fish AND rainbow with the same function
        System.out.println(fish + rainbow);
    }

}
[/java]
```

!Overriding:
- Declaring a method that exists in a superclass again in a subclass, with the same signature
- Methods can only be overriden by subclasses

```java
public class Parmesan extends Cheese {
    public boolean print(boolean rainbow) {
        return rainbow;
    }
}
```

Why override?
- Subclasses can extend functionality from a parent
- Subclasses can override/change functionality from a parent
- Makes the subclass behaviour available when using variables of a superclass

---

Polymorphism and Abstract Classes

!getClass:
- Returns a Class object representing the class of an object

!instanceof:
- Returns true of an object A is an instance of the same class as object B, or a class that inherits from B

!Up Casting:
- When an object of a child class is assigned to a variable of an ancestor class

!Down Casting:
- Whan an object of an ancestor class is assigned to a variable of a child class
- Only makes sene if the underlying object is actually of that class.
- Like I did with player and sprite in Project 1

!Polymorphism:
- The ability to use an object of method in many different ways.
- Means "multiple forms".

Overloading same method with various forms depending on signature
- (AdHoc 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:

- Defines a superclass method that is common to all subclasses, but has no implementation
- Each subclass then provides its own implementation through overriding
- Defines a class that has abstract methods
- Any class with abstract methods must be defined as abstract, but can have zero abstract methods

<Abstract classes are "general concepts", rather than fully realised classes>

Abstract classes cannot be instantiated.

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

```java
public abstract class Robot {
    public boolean move() {
        if (!canMove()) {
            return false;
        }
    }
}

public abstract class AerialRobot extends Robot {
    public boolean move() {
        return super.move() && activateRotors();
    }
}

public class WingedRobot extends AerialRobot {
    public void move(...) {
        if (super.move()) {
            setMotorSpeed(...);
            return true;
        }
    }
}
```

In the above cases, Robot and Aerial Robot cannot be instantiated
i.e. (used with new)

but Winged Robot can be instantiated.

gg midsem

!Interface:
- Works a lot like an abstract class, except <only consists of constants and method signatures>
- Usually used to define a likeness of an entity
- All methods implied to be abstract
- All attributes implied to be static final
- <All methods and attributes are implied to be public>

- Can be extended like classes (by other interfaces)

// the journey continues

!Sorting
- Done via a .sort() method
- In the Comparable<T> class
- Comparable<T> classes must implement int compareTo(object)

There is also a thing about Inheritence vs Interface - when do you use abstract classes as inheritance and when do you use interfaces?

Inheritence specifies a "is a" relationship i.e.

-> Water "is a" liquid
-> Sam "is a" failure
-> Java "is a" not fun language

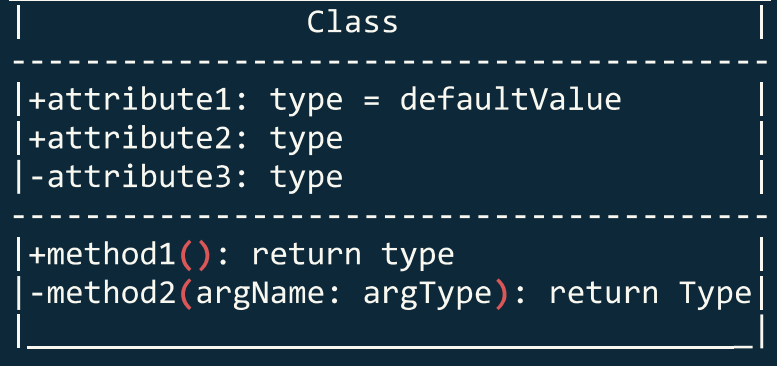
Interface specifies a "can do" relationship

-> Characters in a game "can" talk to the player
-> Students "can" fail
-> Programs "can" segfault

UML Design

// not being able to draw kinda sucks
// not gonna try an ASCII everything, so look at lecture for pictures

In general, we represent a class like this:



Don't need to show the constructor or getters and setters (unless you want to show how only some things have getters/setters)

Key Guide:
    + public
    ~ package   // don't really need to remember these two
    # protected
    - private

Components of an attribute:
- Name (e.g. xPos)

- Data Type (e.g. : int)
- Initial Value (e.g. = 0)
- Privacy (e.g. +)
- Static (underlined)
- Multiplicity (e.g. 2 players)
    - multiplicity is to do with data structures, they can be:

Finite	[10]	Array
Range (known)	[1..10]	Array or List, etc.
Range (unknown)	[1..*]	List, etc.
Range (zero or more)	[*]	List,m etc.

Components of a method:
- Name
- Privacy
- Return type // these two are optional sometimes
- Parameters

<Abstract more> if you see yourself putting things like <coordinates> into a class when designing UML.

!Association
A link indicating a class <contains another class> as <an instance variable> or <attribute>

If you want your <class>, say GameObject to have <an instance variable> of <another class> (e.g. position), then use a <black solid arrow> towards the instance variable. ----->

Directionality:
An association link can be unidirectional (hierarchical, ownership) or bidirectional (co-operation)

Role Labels:
Describing what the relationship is (e.g. "located at")

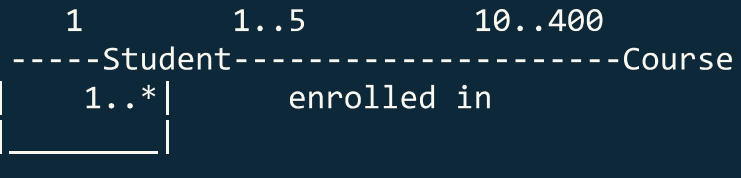
Multiplicity:
Association labels also have multiplicity showing the number of instances of the attribute

e.g.

says GameObject is located at 1 possible position

!Self-Association:
When a <class has an instance variable of itself>

Draw a rectangly thing around the class



!Aggregation:
This is different where one class "has" another class, but both exist independently

This is signified by an empty diamond. <>-------

e.g.

Pond<>------------------Duck

If the Pond class is deleted, the Duck class still works by itself.

!Composistion:
This is the <opposite of aggregation>, where all the <components must make up> a class or entity

This is signified by a filled in diamond. <▪>----------

University<▪>-------------Department<>--------------Professor

So if the university class is deleted, the department class will no longer work. But, the professor class will work regardless.

!Inheritence:
Like an association, but the subclass <inherits> all the superclass' properties

This is signified by a empty arrow. <|----------

!UML Abstract:
Italicised methods or classes are abstract.
// note: in the exam, just use a legend or symbol or something

!UML Interface:
Note: When <inheriting from interfaces>, you must <use a dashed line>
These are signified by a "<<>>", e.g. "<<interface>> Targetable"

Generics

// I skipped the most of the first part of generics because reasons
// whooo frameworks and collections
// If you're confused, go look up the javadocs about collections

!Type Paramaters:
- Some classes have <type parameters>
- Look like this: ArrayList<T>
- <T> can be any class that you want it to be
- Allows us to <define a class or method that uses arbitrary, generic types> that applies to <any and all types>

!ArrayList:

- Class with array as an instance variable
- Can be iterated like arrays (with an Iterator<T>)
- Automatically resizes
- Inserts, removes and modifies elements at any index
- Can be sorted
- Can be toString()'d
- Can't be indexed

[java]
import java.util.ArrayList;
import java.util.Collections;
...
ArrayList<Robot> stuff = new ArrayList<>();
[end]

// Here's a cool use of ArrayList where it continually takes user input
// and makes an ArrayList of it:
[java]
public ArrayList<Integer> generateList(Scanner scanner) {
    ArrayList<Integer> numbers = new ArrayList<>();
    while (scanner.hasNextInt()) {
        numbers.add(scanner.nextInt());
    }
    return numbers;
}
[end]

The collections framework has a bunch of <different data structures> you can use,
including <LinkedLists, Sets, PriorityQueue etc>.

<All of these structures have common operations>
Length   int size()
Presence   boolean contains(Object element)
Add   boolean add(E element)
Remove   boolean remove(Object element)
Iterating   Iterator<E> iterator()
Iterating for   (T t : Collection<T>)

!HashMap:
Another cool data structure is a <HashMap>.
A hash map is similar to a hash table or a dictionary in Python.
e.g.
[java]
// To make a HashMap, call it the same way you would with ArrayList
// HashMap<Key, Value> map = new HashMap<>();
// If we were to make a phonebook:

HashMap<Integer, Person> phonebook = new HashMap<>();
// for project 2 you could use (and you did) a hashmap to store the world
[/java]

!PriorityQueue:
A priority queue is a queue that <orders it's elements> according to the
<specified comparator>.

e.g.
[java]
```
PriorityQueue<Integer> descending =
new PriorityQueue<>(new Comparator<Integer>() {
    @Override
    public int compare(Integer arg0, Integer arg1) {
        return arg0-arg1;
    }
    });

descending.addAll(numbers); // assume numbers is an ArrayList of integers
// You can add all elements to a collection with this method
// Note it only works with collections

// You can even do things like compare lengths of strings!
// Videogame inventory ordering anyone?

PriorityQueue<Integer> shortest =
new PriorityQueue<>(new Comparator<Integer>() {
    @Override
    public int compare(Integer arg0, Integer arg1) {
        return Integer.toString(arg0).length()
        - Integer.toString(arg1).length();
    }
    });

shortest.addAll(numbers);

// what about first character?

PriorityQueue<Integer> ascendingFirstDigit =
new PriorityQueue<>(new Comparator<Integer>() {
    @Override
    public int compare(Integer arg0, Integer arg1) {
        return Integer.toString(arg0).charAt(0)
        - Integer.toString(arg1).charAt(0);
    }
    });

ascendingFirstDigit.addAll(numbers);
```
[/java]

Elements of a PriorityQueue <are not iterated in order>, so to get around this you need to use <.poll()>.

[java]
```
public static void printQueue(PriorityQueue<Integer> queue) {
    Integer[] nums = new Integer[queue.size()];
    int i = 0;
    while (i < queue.size()) {
        nums[i] = queue.poll();
        // .poll() gives you the smallest value in the queue!!
    }
```

```
    System.out.println(Arrays.toString(nums));
}
[/java]

!Anonymous Class:
An anonymous class is when a class is created in the middle of a file
"on the fly" and has no file or class name.
# Note
Anonymous classes/functions are <very very niche>. Don't use them in the project


!Comparator:
A comparator is an object that compares two things.
These two things have to be <Comparable> which <allows objects to be compared>
<PriorityQueue is built> on <Comparators>


!Generics:
Generics give us <flexibility>; code once, reuse the code for <any type>.
If not for generics, we'd have to write Objects for every type ever.

e.g. Generic Methods:
[java]
public <T> int genericMethod(T arg);
public <T> T genericMethod(String name);
public <T> T genericMethod(T arg);
public <T, S> T genericMethod(S arg);
[java]

e.g. Write a generic method that accepts two arguments:
- Array: an array of unknown type
- Item: an object of the same type as the array
And checks for the frequency of said item in array.
[java]

public static <T> int freq(T[] arr, T arg) {
    int count = 0;
    for (T arrayItem : arr) {
        if (arg.equals(arrayItem)) {
            // we also need to override the .equals() method
            count++;
        }
    }
    return count
    // note that we should also account for null being in the type
}

[java]
Note that this will not work for primitive types, ONLY classes.

<Note that if you have a generic class>, you won't need to put the symbols
around each <T>, as the entire class deems T as <T>.
```

e.g.

```java
public class GenericClass<T, U, V> {
    public T genericMethodOne() { // Note how T is without <T>
        //...
    }
    public double genericMethodTwo(U u, V v) { // but function args are the same
        //...
    }
}

// implement a couple class which couples two things together

public class Couple<A,B>{
    private A thing1;
    private B thing2;
    public Couple(A thing1, B thing2) {
        this.thing1 = thing1;
        this.thing2 = thing2;
    }
}
```

```java
// also the iphone x is weird - matt de bono probably
```

Sometimes we need to <guarentee a class' behaviour>.
We can apply <bounds to type parameters>.
This is kind of like "deriving... " in Haskell if you've done Declarative

```java
// looks like this
public class Generic<T extends <class, interface...>> {
}

// actual usage

public class Generic<T extends Comparable<T>> {
}
public class Generic<T extends Robot> {
}
public class Generic<T extends Robot & Comparable<T> & List<T>> {
}

// data Generic T = T (deriving Robot, (Comparable T), (List T))

// Note, you must ALWAYS STATE CLASSES BEFORE INTERFACES if you have multiple
// extends
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }
interface D { /* ... */ }

class BoundedTest<T extends B & C & D> { /* ... */ } // Correct
class BoundedTest<T extends A & B & C> { /* ... */ } // Correct
```

```
class BoundedTest<T extends B & A & C> { /* ... */ }  // compile-time error
//

[/java]

# Generic programming is powerful, HOWEVER!!:
You <CANNOT> instantiate parametrized objects and you can't make arrays of
parametrized objects.

This is the result of <compilation>, where java does a thing and deletes T

You CANNOT do <new> on a generic type.

# DO NOT DO THIS!!!!
[java]

T item = new T();
T[] elements = new T[];

[java]

// e.g.:
// write a class called tracker which has two type parameters,
// the first type must be subclass of Person, and second of Locator
// A Person obbject could be a Hiker, Diver or Pilot.
// A Locator object could be GPS, Infrared or IP.
[java]
public class Tracker<<Person extends Hiker & Diver & Pilot>,
<Locator extends GPS & Infrared & IP>> {
    Person person;
    Locator locator;

    Tracker(Person person, Locator locator) {
        this.person = new Person();
        this.locator = new Locator();
    }
}
[/java]

Equality:
== compares <references> for equality, .equals compares the attributes.

What does it mean for a class to be immutable?:
Immutability:
Can't be modified after being created.

What is a privacy leak?:
Privacy Leak:
A outside/external class has a reference to and can modify the attributes of a
private instance variables of a class.

What is the downside of using protected visibility?:
Protected: Variables can be freely accessed by subclasses
The whole point of privacy is that <we restrict access from outside the class>
```

so it's useless to use protected.

Super:
Used to reference an object's super class

Exceptions

!Errors:

Tyes of Errors:
- Syntax:
Written illegal code, <won't compile> and won't run
(e.g. missing a semicolon)

- Semantic:
Program compiles and runs but the <output is incorrect>

- Runtime:
An error that causes your program to crash and <close prematurely>
(e.g. division by zero)

How to fix Errors:
Solution 1: Do nothing (really bad idea)
Solution 2: Use guards/cases to stop invalid situations (defensive programming)
Solution 3: Exception handling!

Exception handling is all about trying code and then catching the error statements

!Exception:
Error state created by a runtime error in code.
All exceptions <are objects>. They all <have descriptions of the problem>.

Exceptions look like this:
[java]
```java
public void method(...){
    try {
        // attempt some code
        // if an error occurs, go straight to catch block
    }
    catch(<ExceptionClass> varName) {
        // do this if something goes wrong
        // e.g. fixing an index or adjusting input
        // you can chain these
        // If there is another error here, you need to wrap it in another try
        // catch block
    }
    finally {
        // do this whether you fail or not
        // do clean up like closing files
```

```
    }
};
[/java]

You can also <throw an exception>. You need to define so in the method as
accordingly

[java]
public double divide(double a, double b) throws ArithmeticException {
    if (b==0) {
        throw new ArithmeticException();
    }
    return a/b;
}
[/java]

The above does not solve anything, it just complains that it has an exception

// Slick does this for everything and it's real annoying
// bad Design

Note that you should only use throw when you are certain an error will be thrown


<Exceptions are classes!>
- All exceptions <inherit from an Exception class>
- We can define our own exceptions.
[java]
public class JokeISBadException extends Exception {
    public JokeISBadException(String message) {
        super(message);
        // This is an error with a specific case message
    }
    public JokeISBadException() {
        super("git gud lol");
        // This is just a general error that is called if there is no case
    }

}
[java]

You can <chain exceptions>. The <exception specified first> is of highest
priority
This means you can put a "catch (Exception e)" "at the <end of your chain> to
account for <all exceptions>.

[java]

...
} catch (ArithmeticException e) {
    // arithmetic exception
    } catch (ArrayIndexOutOfBoundsException e) {
    // out of bounds
    } catch (Exception e) {
```

```
    // catches literally any other possible error
    // e.StackPrintTrace();
}
[java]

--------------------------------------------------------------------------------

                                Testing

--------------------------------------------------------------------------------

//Code Formatting is pretty self explanatory, I'm not going to write it here
//since everyone has different styles and you should do you. :^)

!Documentation:
//comment styling blah blah
Javadocs!
Javadocs are very cool. They compile to HTML and work as a good reference for
your code. Oracle has Javadocs for every thing they do:
# https://docs.oracle.com/javase/8/docs/

Python has something similar called docstrings

!Software Design:
Bad things:
- Rigidy:
Hard to modify - if you want to change something you need to change a lot
of things
- Fragility:
Changing one part of the system causes unrelated parts to break
- Immobility:
Cannot decompose the system into reuseable modules
- Viscosity:
Writing hacks when adding code in order to preserve the design
- Complexity:
Premature optimization is dumb
Don't make it harder for yourself
- Repetition:
Looks like it's cut/paste
- Opacity:
Lots of convoluted logic, design is hard to follow

GRASP:
- General
- Responsibility
- Assignment
- Software
- Patterns/Principles

!Cohesion:
Classes are designed to solve clear, focused problems.
All methods/attributes contribute to that purpose.

!Coupling:
```

The degree of interaction between classes; invoking another class' methods or accessing/modifying its variables.

!Open-Closed Principle:
Classes should be <open> to extension, but <closed> to modification

!Abstraction:
Solving problems by creating abstract data types to represent problem components achieved in Java through classes, which represent data and actions.

!Encapsulation:
Details of a class should be <hidden> or <private>

!Polymorphism:
The ability to use an object or method in many different ways

!Delegation:
Keeping classes focused by passing work to other classes
e.g. not doing everything about the player in world class

!Unit Testing:
Verifying the operation of a unit by testing a use cases (input/output)

!Manual Testing:
Testing code in an ad-hob manner (the way you'd test prolog queries kindof)

!Automated Testing:
Like that time in project 1 declarative where we used that average.py

!TestCase Class:
A class dedicated to testing a single unit

!TestRunner Class:
A class dedicated to executing the tests on a unit

[java]
```java
// so this is this testcase class, which is dedicated to testing a single unit
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);
    }
}

// And this class runs the test
public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(BoardTest.class);

        for (Failure failure : result.getFailure()) {
            System.out.println(failure.toString());
```

```
        }

        System.out.println(result.wasSuccessful());
    }
}
[java]

Software Tester: Conducts tests on software, primarily to find/eliminate bugs

---------------------------------------------------------------------------

                                Design Patterns

---------------------------------------------------------------------------

Gotta make a <modular> design.
Without a design pattern we'll probably end up with all the issues of
Rigidity, Opacity, Fragility, Immobility, Viscosity, Complexity, Repetition etc.

!Factory Pattern:
In this pattern, <Creators> create <Products>
- <Delegates> object creation to subclasses
- <Abstracts> object creation by using a factory (object production) method
- <Encapsulates> objects by allowing subclasses to determine what they need

!Creator:
An abstract class that <generalises the objects that consume and produce objects>
from the factory; generally have some operation (e.g. the constructor) that
will invoke the factory method

!Product:
An abstract class that <generalises the things being created/produced> in the
factory

// approach in project 2 was really similar to this - GameManager/Loader class
// created list of sprites
[java]

public abstract class Game {
    // we do not know what players we need yet
    private final List<Player> team = new ArrayList<>();

    // separate logic of creating game with deciding which player we need
    public Game(int nPlayers) {
        for (int i = 0; i < nPlayers; i++){
            Player player = createPlayer();
            team.add(player);
        }
    }

    abstract protected Player createPlayer();
}

// subclasses can be adapted to many different types of games depending on
```

```java
// what the devs need whenever they start a new project
public class RPGGame extends Game {
    @Override
    protected Player createPlayer() {
        return new RPGPlayer();
    }
}

public class FPSGame extends Game {
    @Override
    protected Player createPlayer() {
        return new FPSPlayer();
    }
}

RPGGame unimelbCSDatingSim = new RPGGame();
FPSGame callOfSegFault = new FPSGame();

[/java]
```

How do you define a design pattern?:
It is a <template that has a structure/design> that allows:
- <Reuseable> Solutions
- <Applicable> to many problems
- General Guidelines
- Provided as a <template>
- "Best Practise" solutions

!Analysing a Pattern:
- **Intent:**
  The goal of the pattern, <why does it exist>
- **Motivation:**
  A <scenario/example> where this pattern can be used
- **Applicability:**
  General situations where you can <use the pattern>
- **Structure:**
  <Graphical representations of the pattern>, likely a UML diagram
- **Participants:**
  List of class/objects and their <role in the pattern>
- **Collaboration:**
  How <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

```
// in the exam he MAY give you a NEW design pattern
// need to discuss/analyse of how it works, potential downsides etc.
```

!Observer Pattern:
An <observer> looks at another class and <responds to the given class>.

Once there is <a change in the observed class> the <observer changes>.

The observer pattern is really nice because it <usually decouples the classes>
while <increasing cohesion> (less information shared between classes).
It is also very <flexible> as you can <add and remove observers at will>.

This is also called publish-subscribe model

!Subject Class:
The observed class, e.g. a player

!Observer Class:
The object which monitors the subject <in order to respond to> it's state,
e.g. an AI enemy

[java]

```
import Java.util.Observer;

// Enemy implements Observer

public class Pointer extends Enemy {
    public update(Observable o, Object arg) {
        // go to pointer and kill player
    }
}

public class Bug extends Enemy {
    public update(Observable o, Object arg) {
        // put a "bug" on the player
    }
}

public class Player implements Observable {
    public Player(..., ArrayList<Observer> observers) {
        ...
        // iterate through observers
        for (Observer o : observers) {
            this.addObserver(o);
        }
        notifyObservers("Player created");
    }
}
```

[/java]

// a lot more design patterns will be learnt in SWEN30006

There are usually 3 types of design patterns
// don't need to know this
- creational, structural and behavioural

!Sequential Programming:
"Top to bottom" programming. Start at the top and end at the bottom.
- Useful for static programs

!Event Programming:
Programming dependent on the <state> of objects. Whenever a <state is altered>, an <event> is recorded. A <callback> then <responds to> the <event>.
Event-driven programming is using events and callbacks to control the flow of a program

e.g.
Exception Handling and the Observer P. is an example of event-driven programming

One use case in Slick is keyPressed();
// case study
[java]
```
public class Player implements KeyListener {

    @Override
    public boolean isAcceptingInput() {
        return true;
    }

    @Override
    public void keyPressed(int key, char c) {
        // lets us respond to key presses without needing
        // to access the input object directly
    }
}

public class World {
    public void addListeners(GameContainer gc) {
        Input input = gc.getInput();

        input.addKeyListener(player);
        input.addKeyListener(rogue);
    }
}
```
[/java]

Pros of event-driven programming:
- Improves performance of code by not having to constantly check for events
- Don't have to parse the input, therefore <better cohesion with less coupling>
- Improve <encapsulation>
- <Avoid> World having to explicitly <send> the player information <about input>
- Easily add/remove <behaviour to classes>
- Easily add/remove <additional responses>

Some examples of <events>:
- Controller/Keyboard/Mouse Input
    - GUI/Movement/Action

- Time
    - Timed challenges etc.

Cons of the while(true) + if statements Event Loop:
- Polling - program <actively enquiring> about the state of something
- Lots of waiting
- Always <responds in the same order>
- Can't "escape" from one method to respond to something else

!Asynchonous Programming:
Programming with <interrupts> which is a signal generated by hardware/software
indicating and event that needs <immediate CPU attention>
The execution of the program is <out of order> and <not in the dev's control>

Interrupt Service Routine:
Equivalent of callback in event driven programming

Interrupts are <very low level>, IMMEDIATELY take control of the program
e.g. Exceptions, activating sleeping process/task, device drivers

an interrupt looks like this:
[java]
```java
// this works in a separate thread to the main program
public class DistanceSensor {
    public void detectCollision() {
        while(true) {
            if (...) {
                // if a collision is detected, alert the main thread
                mainThread.interrupt();
            }
        }
    }
}
```
[java]

!Inheritance-based Game Design:

- Most <obvious way> to design a game in a OO language is to <use inheritance>.
- Entity <abstract base class> to represent game objects, <inherit from> entity
- Use <polymorphism>
- Use the <factory pattern>

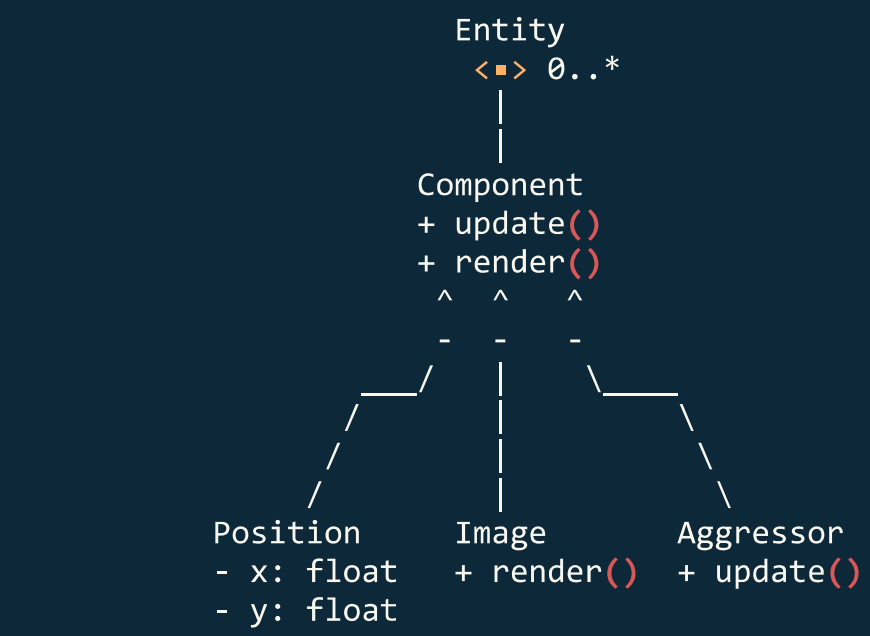
// basically project 2

Problem: Sometimes we want things that can inherit from two types
(e.g. passive and aggresive AI that changes state)
- Double inheritance (doesn't exist in java)

!Composition over Inheritance:
You can break down each object into it's key components, for example
an entity then becomes a composition of it's components:

a player can be moveable, pusher, controllable etc.

Usually the structure will be as following:
A entity class will be the super class for several different component subclasses
e.g.



Here we can <override any of the subcomponents> and <adapt to any situation>.

To make a player or a enemy, just create an <instance of entity>
-> Iterate through components and render and update accordingly

This is called the entity-component approach

Used by the game engine Unity (uses C# which handles generics better)

// However we've basically just made C structs, breaking encapsulation.
// rip oo principles

We can take this approach further and go to the entity-component-system approach
A system updates <all the entities> with the specified components.

e.g. a <physics system> updates <all entities affected by gravity>

This is how most <AAA> games work, and <we're not really doing Object Oriented>
anymore

// exam: need to define sequential, event-driven, asynchonous
// examples of event systems, MUST BE ABLE TO DO INTERRUPTS!
// entity-component isn't really examinable

Advanced Java & Object Oriented Programming

// nearly there guys
// just hang in there
// save the ult

!Enumerated Types:
A class that consists of a <finite list of constants>.
e.g. to represent some cards we'd have all the faces and numbers possible

[java]

```java
public class Card {
    public Rank rank;
    public Suit suit;
    public Colour Colour;

    public Card(Rank rank, Suit suit, Colour colour) {
        this.rank = rank;
        this.suit = suit;
        this.colour = colour;
    }

}

public enum Rank {
    // this is the highest 'index'
    ACE,
    TWO,
    THREE,
    FOUR,
    FIVE,
    SIX,
    SEVEN,
    EIGHT,
    NINE,
    TEN,
    JACK,
    QUEEN,
    KING
    // this is the lowest 'index'
}

// these should be tied together
public enum Colour {
    RED, BLACK
}

// we can tie suit to colour like this:
public enum Suit {
    SPADES(Colour.BLACK),
    CLUBS(Colour.BLACK),
    HEARTS(Colour.RED),
    DIAMONDS(Colour.RED),

    // you can do all sorts of computations after
    private Colour colour; // this can also be public final

    Suit(Colour colour) {
```

```
        this.colour = colour;
    }
}

Rank rank = Rank.ACE; // these are all constants so we just use .ACE
Card card = new Card(Rank.FOUR, ..., ...);

// if we have an ArrayList of rank like so:
ranks.add(Rank.TEN);
ranks.add(Rank.ACE);
ranks.add(Rank.THREE);
ranks.add(Rank.KING);

Collections.sort(ranks);
// it will sort in ascending order. We can change by defining a comparator.
// the enum types are also ints, so just compare them with quik maffs

[java]

These values are <accessed statically> because they are constants.

Enums come pre-built with:
- Default constructor
- toString()
- compareTo()
- ordinal()

we can also override any of these if we want:

[java]
// defined in Rank enum
public boolean isFaceCard() {
    return this.ordinal() > Rank.TEN().ordinal();
}
[java]

Enums are also immutable.

// what is an enum?
// Describe an enum, and give an example of it:
// an enum is an enumerated type, which is a class which can only be
// a finite list of constants
// e.g. Rank in Mario Kart
--------------------------------------------------------------------------------
!Functional Interfaces:
An interface which only contains a Single Abstract Method

[java]
@FunctionalInterface
public interface Attackable {
    public void attack();
}
[java]

```

Functional interfaces are pretty weird, but we can do things like:

!Predicate:

```
public interface Pedicate<T>
```

Predicate<T> is a cool one, it is a <functional interface> which represents the functionality of calculating a boolean value or <applying a test to something>

- Executes a boolean test(T t) method on a single object

```
// This is a bit like prolog, how every function is a true/false test
// having the predicate functional interface just means the type can be tested
// for true/false
```

e.g. want to check if a string is uppercase, so we implement predicate

!UnaryOperator:

UnaryOperator<T> is another functional interface which modifies or applies a method or operation to T.

```
// used in mapping

// exam question: give a specific example of how you might use a
// ToIntFunction<T> functional interface?
// - hashing function which converts whatever to int
// - converts a worded version of a number to an actual int
```

!Lambda Expressions:

```
// hey Haskell and Python
```

A technique that <treats code as data> that can be <used as an object> and allows us to <instantiate an interface> without implementing it.
e.g.

```
// this checks for whether an integer is larger than zero
// we can make the entire functional interface in one line without having to
// do override etc.
Predicate<Integer> p = i -> i > 0;

// syntax is
// (sourceVariable1, sourceVariable2, ...) -> <operation on source variables>
[java]
// e.g. Filtering a list using predicate

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, 3, 4, 5, 6, 7, 8);
List<Integer> filteredList = new List();

for (Integer i : nums) {
    if (p3.test(i)) {
        filteredList.add(i); // add to new filtered array
    }
```

```
}

// e.g. replacing a value in a list (like map)
public abstract class List<T> {
    public void replaceAll(UnaryOperator<T> operator);
}

List<String> names = Arrays.asList("Jon", "Arya", "Danaerys");

names.replaceAll(name -> name.toUpperCase());
System.out.println(names);
[java]

<Lambda Expressions> can be used <in place of anonymous classes> but are
NOT THE SAME THING!

[java]
// anonymous class vs lambda expression
// anonymous class
starWarsMovies.sort(new Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {
        return m1.rating - m2.rating;
    }
)};

// lambda expression (so much nicer)
starWarsMovies.sort((m1, m2) -> m1.rating - m2.rating);
// java usually can figure out the types of m1 and m2, we can define it
// ourselves if we want: (Movie m1, Movie m2) -> ...
[java]

// So basically, UnaryOperator<T> is similar to map in Haskell,
//               Predicate<T> is similar to filter in Haskell

!Method References:
[java] names.replaceAll(String :: toUpperCase()); [java]

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>.

[java]
// lambda
UnaryOperator<String> upper = name -> name.toUpperCase();

// method ref
UnaryOperator<String> upper = String::toUpperCase;

// Static methods:
Class::staticMethod
Person::printWarning

// Instance methods:
Class::instanceMethod || object::instanceMethod
String::startsWith || person::toString
```

```
Class::new
String::new
[java]

Method arguments are <implied> in method references when they're called.

[java]
// the ultimate showdown - check if a number is odd using a
// specified Numbers class
// 4v4 last 8 in pubg last circle let's go

// Using an anonymous class
findNumbers(list, new Predicate<Integer>() {
    public boolean test(Integer i) {
        return Numbers.isOdd(i);
    }
});

// lambda functions
findNumbers(list, i -> Numbers.isOdd(i));

// Method Reference
findNumbers(list, Numbers::isOdd);
[java]

!Streams:
A series of elements given in squence that are <automatically> put through a
<pipeline> of operations (like map)

[java]
// 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.

// First, we have basically 5 steps:
// Iterate through list
// Select elements w/ length greater than 5
// Select elements with first character "C"
// Converting it to uppercase
// Adding element to list

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

// damn, that was neat
[java]

// Eyy, map and filter make an appearance!
// streams are fuckin cool
```

Streams have a bunch of sequential 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 to list, array, String)

[java]

```java
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);

String output = list.stream()
    .filter(i -> i%2==0)                // filter out odd
    .map(i -> new Integer(i*i))         // note you have to do new
    .filter(i -> i >= 100)              // filter out less than 100
    .map(Integer::toString)             // don't forget to convert to string!
    .collect(Collectors.joining(", ")); // join
```

[java]

!Variadic Methods:

A method that takes an unknown number of arguments. It can adapt accordingly.

The way to achieve this is really similar to Haskell's list comprehension

[java]

```java
// Note: Variadic methods convert input args into an array, which can be
//          dangerous
public String concatenate(String... strings) {
    String string = "";

    for (String s : strings) {
        string += s;
    }

    return string
}
```

[java]

```java
// You WILL NOT be expected to write any of the above code in this section,

// BUT you should be able to read, explain and understand it



// we did it team
// we won
// gg

// godspeed     o7
```