1 Basic Commands

identifier: a name that uniquely identifies a program element such as a class, object, variable or method
data types: two main data types, primitive and non primitive
primitive: intrinsic data type, eg. `int, float, char, bool`
non-primitive derived data type, eg. `class, array, interface, object, string`
two-way statement: combines a requirement with two options
eg. `flag = (i<10) ? 0:1;` , if `i=5` then `flag = 0`
`import:` adds library, similar to `#include` in c
`final:` declares a constant, can be declared then assigned later, similar to #define
polymorphism: being able to use objects or methods in different ways
overloading: same method with various forms depending on signature
overriding: same method with various forms depending on class
substitution: using subclasses in place of superclasses
generics: defining parametrised methods/classes

2 Classes

class: a fundamental unit of abstraction in OOP, represents an entity that is part of a problem
object: a specific, concrete example of a class
instance: an object that exists in your code
data abstraction: creating new data types that are well suited to an application by defining a new class
encapsulation: grouping data(attributes) and methods that manipulate the data to a single entity through defining a class
instantiate: to create an object of a class, use keyword `new`
constructor: a method used to create and initialise an object
`this:` a reference to the calling object, ie. the object/class that owns the method
method overloading: methods that have the same name, but are distinguished by their signature
polymorphism: processing objects differently depending on their data type or class

static members: methods and attributes that are not specific to any object of the class
static variable: a variable shared amongst all objects of the class, eg. if count is in circles, then every circle can access it
static method: a method that doesn't modify or access any instance variables of a class, can only call other static methods, and access static data only (no class data) *be careful when using static methods*
instance variable: only one copy per object, eg. `circle.radius` will be different for each instance
copy constructor: a constructor that takes the input of an object, and creates a new instance of it with the same values. Different to assigning it to an object, as it will have a new reference.
`equals:` compares if two objects are equal, (`==` compares if two references are equal)
package: allows to group classes and interfaces into bundles, similar to libraries
To place a class in a package, `package <directory_name>` + class code

information hiding: the ability to 'hide' the details of a class from the outside world
`public:` makes it available/visible everywhere (within the class, and outside the class)
`private:` makes it only visible to within the class, (cannot be inherited, or visible by subclasses)
`protected:` makes it visible within the class, subclasses and within all classes that are in the same package as that class
access control: preventing an outside class from manipulating the properties of another class in undesired ways
mutable: a class that contains a method that can change the instance variable is called mutable
immutable: a class that contains no methods (aside from constructors) that can change any of the instance variables
wrapper classes: a class that gives extra functionality to primitive data types, and lets them behave like objects
parsing: processing one data type into another
unboxing: the process of converting a primitive to/from its equivalent wrapper class

3 Input and Output

command line argument: information or data provided to a program when it is executed, accessible through the `args` variable
scanner: importing the scanner `import java.util.Scanner;`
create *one* scanner for each program `Scanner scanner = new Scanner(System.in);`
`System.in:` an object representing the standard input stream, or the terminal/keyboard
`catch:` acts as a safeguard for potential errors, prints an error message if anything is wrong

4 Inheritance and Polymorphism

inheritance: a form of abstraction that permits 'generalisation' of similar attributes and methods of classes, analogous to passing genetics on to your children, *allows for code to be reused*
superclass: the 'parent' or 'base' class in the inheritance relationship; provides general information to its 'child' classes
subclass: the 'child' or 'derived' class in the inheritance relationship; inherits common attributes and methods from the 'parent' class
`extends:` indicates one class inherits from another
`super():` invokes a constructor in the parent class, used like a method
`@override:` when the child class method overrides the method in the parent class, the annotation is optional
overriding: declaring a method that exists in a superclass again in a subclass, with the same signature. Methods can only be overriden by subclasses
overloading: declaring multiple methods with the same name, but differing method signatures. Superclass methods can also be overloaded in subclasses
`super.CLASS:` also a reference to an object's parent class, similar to `this`, but referring to the attributes and methods of the parent
`final:` Indicates that an attribute, method or class can only be assigned, declared or defined once. Used if you don't want subclasses to override a method.
`public:` makes it available everywhere
`protected:` makes it visible only within the class, subclasses and within all the classes in the same package. Good for methods if accessed by child classes, but generally *don't* assign to attributes.
shadowing: when two or more variables are declared with the same name in overlapping scopes, ef. a subclass and a superclass. In general, just best to avoid.
`getClass:` returns an object of type Class that represents the details of the calling object's class
`instanceof:` an operator that gives `true` if an object A is an instance of the same class as object B or a class that inherits from object B
upcasting: when an object of a *child* class is assigned to a variable of an *ancestor* class, will always be valid ie. piece p = new rook, p is both a rook and a piece
downcasting: when an object of an *ancestor* class is assigned to a variable of a *child* class, onlt makes sense if the object is actually of that class. Generally only use for .equals methods
polymorphism: the ability to use objects or methods in many different ways; meaning multiple forms eg.(overloading, overriding, substitution, generics)
substitution: using subclasses in place of superclasses
abstract class: a class that represents common attributes and methods of its subclasses but is also missing some information specific to its subclasses. Cannot be instantiated (i.e can't create an instance of type piece)
concrete class: any class that is not abstract, and has well-defined, specific implementations for all actions it can take
abstract method: defines a superclass method that is common to all subclasses, but has no implementation. Each subclass would provide it's own implementation through overriding (empty method)

5 Interfaces and Polymorphism

interface: declares a set of constants and/or methods that define the *behaviour* of an object, similar to a can do relationship, generally called ¡...¿able
implements: declares that a class implements all the functionality expected by an interface
default: incidates a standard implementation of a method that can be overridden if the behaviour doesn't match what is expected of the implementing class

6 Generics

type parameter: `T` is a *type parameter* or a type variable, the value of `T` is literally a type, and when `T` is given a value, every instance of the placeholder variable is replaced (Comparable interface for example)
`ArrayList:` a class that can be iterated like arrays, automatically handles resizing, has a `ToString()` method, and a `compareTo()` method that should be overriden
generic class: a class defined with an arbitrary type for a field, parameter or return type
generic subtyping: generic classes or interfaces are *not* related merely because there is a relationship between their type parameters, eg. cannot assign dog = animal, or animal = dog as errors will occur
wildcard ?: stands for *unknown* type, generally used with when reading from and inserting to a generic collection, eg can print lists of array dogs and bears using similar functionality
extends wildcard: `List <? extends A >` means that a list of objects are instances of the class A or subclasses of A
super wildcard: `List <? super A>` means a list of objects that are instances of class A or superclasses of A
subtyping: `<:` denotes *is a subtype of*
generic method: a method that accepts arguments or returns objects of an arbitrary type

7 Collections and Maps

collections: a framework that permits storing, accessing and manipulating lists (an ordered collection)
maps: a framwork that permits storing, accessing and manipulating key-value pairs
anonymous inner class: a class created "on the fly" without a new file, or class name for which a single object is created
`ArrayList:` similar to arrays, but better with additional functionality like dynamic length
`HashSet:` ensures elements are unique and has no duplicates
`PriorityQueue:` allows you to order elements in non-trivial ways
`TreeSet:` fast lookup/search of unique elemets
`HashMap:` takes two types, K and V (key and value)

8 Exceptions

Syntax: errors where what you write isn't legal code; identified by the editor/compiler
Semantic: error when the code runs to completion, but results in incorrect output/operation; identified through software testing
Runtime: an error that causes your program to prematurely crash and burn; identified through execution eg. dividing by 0, out of bounds
Exception: an error state created by a runtime error in your code; an exception
Exception: an object created by Java to represent the error that was encountered
Exception Handling: code that actively protects your program in the case of exceptions
`try:` attempts to execute some code that may result in an error state or exception
`catch:` deals with the exception, eg. asking the user to input again, adjust an index or output and enrror message and exit
`finally:` performs clean up, eg. closing files, assuming the code didn't exit

throw: responds to an error state by creating an exception object, either already existing or one defined by you
throws: indicates a method has a potential to create and can't be bothered to deal with it, or that the exact response varies by application
unchecked exception: inherits from the `error` class, can be safely ignored by the programmer, most inbuilt Java exceptions are unchecked as you aren't forced to protect against them
checked: inherits from the `exception` class, must be explicitly handled by the programmer in some way, the complier gives an error if a checked exception is ignored
exception handling: use `catch` or `declare` for handling checked exceptions, enclose code that can generate exceptions in a try-catch block or declaring a method may create an exception by using `throws` clause

9 Design Patterns

singleton pattern: ensures that class only has one instance and provides a global point of access to it
template method pattern: essentially inheritance, defines a family of algorithms, encapsulates each one and then make them interchangable
strategy pattern: implements/interface, allows reuse of specific methods
factory: a general technique for manufacturing (creating) objects
product: an abstract class that generalies the objecte being created/produced by the factory
creator: an abstract class that generalises the objects that will consume/produce products; generally have some operation (eg. constructor) that will invoke the factory method
subject: an "important" object, whose state (or change in state) determines the actions of other classes
observer: an object that monitors the subject in order to respond to its state and any other changes made to it
creational design patterns: solutions related to object creation, ie singleton and factory method
behavioural design patterns: strategy, template method, observer

10 Testing

GRASP: General, Responsibility, Assignment, Software, Patterns/Principles. A series of guidelines for assigning responsibility to classes in an object-oriented design; how to break a problem down into modules with a clear purpose.
cohesion: Classes are designed to solve clear, focused problems. The class' methods/attributes are related and work towards this objective. Designs should have maximum cohesion.
coupling: The degree of interation between classes, dependency between classes. Designs should have minimum (low) coupling.
open-closed principle: Classes should be open to extension, but closed to modification.
abstraction: Solving problems by creating abstact data types to represent problem components; achieved in OOP through classes, which represent data and actions.
encapsulation: The details of a class should be kept hidden or private, and the user's ability to access the hidden details is restricted or controlled. Also known as data or information hiding.
polymorphism: The ability to use an object or method in many different ways; achived in Java through *ad hoc* (overloading), *subtype* (overriding, substituion), and *parametric* (generics) polymorphism.
delegation: keeps classes focused by passing work to other classes. Computations should be performed in the class with the greatest amount of relevant information.
software testing: Important for reducing costs **unit:** a small, well-defined component of a software system with one, or a small number of responsibilities.
unit test: Verifying the operation of a unit by testing a single use case (input/output), intending for it to fail.
unit testing: Identifying bugs in software by subjecting every unit to a suite of tests.
manual testing: Tetsing code manually in an ad-hoc manner. Generally difficult to reach all edge cases, and not scalable for large projects.
automated testing: Testing code with automated, purpose built software. Generally faster, more reliable and less reliant on humans.
`assert:` a true or false statement that indicates the success or failure of a test case
TestCase class: a class dedicated to testing a single unit
TestRunner class: a class dedicated to executing the tests on a unit
software tester: conducts tests on software, primarily to find and eliminate bugs
software quality assurance: actively works to improve the development process/lifecycle. Directs software to conduct tests, primarily to prevent bugs

11 Asynchronous Programming

sequential programming: a program that is run (more or less) from top to borrom, starting at the beginning of the *main* and concluding at it's end
state: the properties that define an object or device; for example if it is 'active'
event: created when the state of an object is altered
callback: a method triggered by an event
event-driven programming: using events and callbacks to control the flow of a program's execution
polling: also known as sampling, relies on the program to actively enquire aboout the state of an object eg. using if else statements - has disadvantages (set order, can't escape from a method until it is complete, unable to execute two methods at once)
interrupt: a signal generated by hardware or software indicating an event that needs immediate CPU attention
interrupt service routine: event-handling code to respond to interrupt signals, ie. callback for an interrupt signal
composition: an alternative to inheritance, allows reuse of code by giving a component entities. eg. if a mob can be aggressive sometimes but non-aggro othertimes, has a component called aggressive with a state

12 Advanced Java and OOP

enum: a class that consists of a finite list of constants
variadic method: a method that takes an unknown number of arguments
functional interface: an interface that contains only a single abstract method, also known as a single abstract method interface
`Predicate<T>:` accepts one argument of type T, and returns true or false
`UnaryOperator<T>:` accepts an argument of type T and returns an object of the same type T
lambda expression: a technique that treats code as data that can be used as an "object", eg allows us to instantiate an object without implementing it
lambda expression: instances of functional interfaces
method reference: an object that stores a method, can take place of a lambda expression of that lambda expression is only used to call a single method eg. `String::toUpperCase`
stream: a series of elements given in a sequence, that are automatically put through a pipeline of operations