Electronics and
Computer Science

UNIVERSITY OF
Southampton

Coursework Specification

Module:	COMP1202: Programming 1		
Assignment:	Programming coursework	**Weighting:**	40%
Lecturers:	Jian Shi, Heather Packer, Thai Son Hoang		
Deadline:	11/Dec/2020	**Feedback:**	15/Jan/2021

Coursework Aims

This coursework allows you to demonstrate that you:

- Understand how to construct simple classes and implement methods.
- Are able to take simple pseudo-code descriptions and construct working Java code from them.
- Can write a working program to perform a complex task.
- Have an understanding of object-oriented programming.
- Can correctly use polymorphism and I/O.
- Can write code that is understandable and conforms to good coding practice.

Contacts

General programming queries should be made to your demonstrator in the timetabled labs.

Queries about the coursework specification should be made to Jian Shi (Jian.Shi@soton.ac.uk).

Any issues that may affect your ability to complete the coursework should be made known to **Jian Shi** (Jian.Shi@soton.ac.uk) or **Son Hoang** (T.S.Hoang@ecs.soton.ac.uk), ideally before the submission deadline.

Instructions

Late submissions will be penalised at 10% per working day.
No work can be accepted after feedback has been given.
You should expect to spend up to 50 hours on this assignment.
Please note the University regulations regarding academic integrity.

ECS Battle Arena (v1.0.0)

Image from vecteezy.com

Specification

The aim of this coursework is to construct a simple simulation of a battle arena. This arena will admit students and they will form a team to battle different teams of monsters.

You are not required to have prior knowledge in gaming to complete this coursework.

Your task is to implement the specification as written.

In some cases, specific names are defined or specific properties given for classes. This is mainly to ensure that we can easily read your code, and find the parts that are most relevant to the marking scheme. You will not gain marks for deviating from the specification in order to increase the realism of the simulation. In fact, it may cost you marks and make it harder to assess what you have written. In some cases, we wish to know that you can use specific Java constructs, so if it specifies an `ArrayList` then we want to see you use an `ArrayList`. There might be other ways of implementing the same feature but we are trying to assess your ability to use specific constructs. Where the specification is less explicit in the way something is implemented (for example, by omitted what data types to use) you are free to make your own choices but will be assessed on how sensible those choices are. Comments in your code should explain the choices you have made.

An FAQ will be kept on the notes pages of answers that we give to questions about the coursework. If issues are found with the specification it will be revised if necessary. If questions arise as to how certain aspects might be implemented then suggestions of approaches may be made but these will be suggestions only and not defined parts of the specification.

How the ECS Battle Arena works

For this coursework you are expected to follow the specification of the students, teams, guilds, and the battle arena as set out below. This will not correspond exactly to a real student, teams, and definitely not monsters nor battle arena in reality but we have chosen particular aspects for you to model **that help you to demonstrate your Java programming**.

The battle arena simulates the battles between teams of students and teams of monsters. A list of characters participated in our simulation are listed in Table 1. The columns list the base attributes, i.e., hit points, attack, defence, speed, and knowledge points of the different characters.

- **Hit points** – the character's hit points determining the character strength. A character's is *alive* when their hit points is positive. We will often use HP as a short-hand for hit points.
- **Attack** – the character's attacker power. The more attack power a character has, the more damage that the character will deal to the enemy when attacking.
- **Defence** – the character's defence power. The more defence power a character has, the less damage that the character will receive from the enemy when being attacked.
- **Speed** – the character's speed. A character with a higher speed will have the turn to make their move before a character with a lower speed.

- **Knowledge points** – the character's required knowledge points to use their special ability. Notice that knowledge points are only relevant for students. We will often use KP as a short-hand for knowledge points in this document.

We will explain how the base attributes contribute to the characters' ability later.

There are a number of concepts, people, and procedures that contribute to this simulation. For our purposes these include:

Students: The different students that join the Guild and fight against the monsters.

Monsters: The monsters that challenge the student guild.

Teams: A battle in the system happens between two teams, formed by a maximum of 5 characters (students or monsters).

Guilds: A guild is formed by a group of students and student teams can be created from the guild members to battle the monster teams.

Character	Hit Points	Attack	Defence	Speed	Knowledge Points
Computer Science (CS)	7	6	6	6	4
Computer Science with Artificial Intelligence (AI)	6	7	7	5	3
Computer Science with Cyber Security (Cyber)	7	7	5	6	6
Software Engineering (SE)	8	5	8	4	10
Minion	5	5	5	5	--
Boss	8	7	8	7	--

Table 1. Base attributes for different characters

The next sections will take you through the construction of the various required concepts. You are recommended to follow this sequence of development as it will allow you to slowly add more functionality and complexity to your final simulation. The first few steps will provide more detailed information to get you started. It is important that you conform to the specification given. If properties and methods are written in `red` then they are expected to appear that way for the test harness and our marking harnesses to work properly.

Part 1 – The Character and Team classes

Characters

The first class you will need to create is a class that represents a *Character.* The properties for **the** `Character` **class** that you will need to define are:

- `name` – this is the unique name of the character,
- `baseHP, baseAtk, baseDef, baseSpd` – this is the base hit points, attack, defence, speed of the character.
- `level` – the character current level. A character starts from Level `1`. A character's attributes are based on their level and base attributes.

Define these as you think appropriate, and create a constructor that initialises them with the following signature.

`Character(String name, int baseHP, int baseAtk, int baseDef, int baseSpd)`

Define accessor methods

- `getName()` – returns the name of the character
- `getMaxHP(), getAttack(), getDefence(), getSpeed()` – return the maximum hit points, the attack, the defence, the speed for the character (taken into account the base attributes and the character current level) accordingly. The attributes are computed according to the following formula.

  Round(baseAtt * (level^1.2))

  Where `baseAtt` represent the base hit points, base attack, base defence, base speed accordingly. "Round" is the mathematical integer rounding function.
- `getTargetEP()` – returns the required target experience points (**EP**) for the character to gather in the current level to go to the next level. The attribute is computed according to the following formula.

  Round(10 * (level^1.5))

Now define the following additional properties:

- `currentHP` – the current hit points of the character. Initially, this should be the same as the character maximum HP.
- `currentEP` – the current experience points of the character.

Define the accessor and mutator methods

- `getHP(), getEP()` – return the current HP, EP of the character accordingly.
- `increaseHP(int amount), decreaseHP(int amount)` – increases/decreases the current HP of the character. Ensure that the current HP of the character is non-negative and no more than the maximum HP for the character.
- `increaseEP(int amount)` – increases the current EP of the character. If the EP reaches the target experience points, the character will level up. In this case, the level of the character is increased by 1, and the current EP resets to 0. If the character is alive, i.e., their current HP is none 0, it resets to the maximum HP (corresponding to the new character level). Otherwise, i.e., if their HP is 0, it stays as 0 (this case can happen when the EP is gained from being attacked).

Teams

The second class you will need to create is to represent a t*eam*. The properties of a team are:

- `name` – this is the name of the team.
- `members` – the members of the team, this is a collection of characters.

Define `name` as you think appropriate, and use an `ArrayList` to represent the members of a team. Create a constructor `Team(String name)` that initialises them. Implement the following getter and setter methods.

- `getName()` – returns the name of the team
- `Character[] getMembers()` – returns the members of the team as an array of Characters.
- `addMember(Character member)` – add a new member to the team. The method returns an `int` indicates the status of the operation as follows.
  - the member is already in the team, returns `-1`
  - the team is full, i.e., already has 5 members, the member is not added and returns -2
  - returns the size of the team after the member has been added.

Extend the `Character` class with appropriate properties, a setter method `setTeam(Team team)` for setting the team for the character, and a getter method `getTeam()` for returning the current team for the character.

You can have any additional properties and methods for your classes as you wish.

BY THIS STAGE you should have a Character class, and a Team class connected to the Character class.

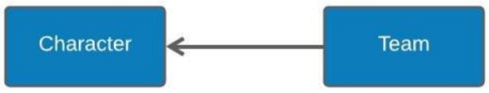


Figure 1. Characters and Teams

You can now test your Character and Team classes by creating a main method and to create some objects of these classes. You can call increaseEP() method on the Character objects and check how their status changes. You can call addMembers() method on the Team objects and check its return status in different situations.

Part 2 – The Different Classes for Students

The first step in this part is to create is a class that represents a *Student*. **The Student class** will be the basis for the students in different programmes. Create a Student class that inherits from the Character class. The properties that you will need to define for *Students* are:

- maxKP – the maximum knowledge points for the student.
- currentKP – the current knowledge points for the student. This must be non-negative and no more than the student's maximum knowledge point. The student can use their special abilities when their knowledge points reach the maximum.

Define these as you think appropriate, and create a **constructor** that initialises them. In particular, define a construct with the following signature.

`Student(String name, int baseHP, int baseAtk, int baseDef, int baseSpd, int maxKP)`

The *Student* class will also need to define some methods:

- increaseKP(int amount) – increases the knowledge point for the character.
- javaProgramming(Character enemy) – uses the Java Programming skill on the enemy. The following actions will be carried out.
    - The character's EP is increased by 3.
    - The character's KP is increased by 1.
    - The enemy's HP is decreased. The damage causes to the enemy will be calculated according to the following formula. Here "/" denotes integer division.

      (100 * student's attack attribute) / (100 + enemy's defence attribute).
    - The enemy's EP is increased by 2.
    - If the enemy is a student, the enemy's KP is increased by 3.
    - If the enemy is deceased, i.e., their HP is 0, then the character's EP is increased by a further 4 points.
- selfStudy() – the student self-studies and will gain 2 HPs, 6 EPs, and 2 KPs.

AIStudents, CSStudents, CyberStudents, SEStudents

To model students in different programmes, you will use inheritance. The base attributes for the different students are in Table 1. Create four classes, namely AIStudent, CSStudent, CyberStudent, SEStudent, all of which inherit from the Student class. For each of these new class, create a constructor that have one parameter for the student's name, e.g. AIStudent(String name).

The different special abilities for the students are as follows. When using any special abilities, the student's EP is increased by 4 and the student KP is reset to 0. For each enemy killed by the student's special abilities, the student gains an additional 4 EPs.

`AIStudent` – The *Artificial Intelligence* students have special ability of using machine learning and natural language processing.

- `machineLearning(Character enemy)` — allows the student to attack an alive enemy and cause double the damage.
  2 * (100 * student's attack attribute) / (100 + enemy's defence attribute).
  The enemy does not receive any EP nor KP.
- `naturalLanguageProcessing()` — allows the student to heal themselves, the amount of HP gained is the student's defence points.

`CSStudent` – The *Computer Science* students have special ability of pair working and support.

- `pairWorking(Character friend, Character enemy)` — allows the student to team up with an alive friend and attack an enemy. The friend and the enemy do not receive any EP nor KP.
- `support(Character friend)` — allows the Student to heal an alive friend, the amount of HP gained is the student's defence points. The friend does not receive any EP nor KP.

`CyberStudent` – The *Cyber Security* students have special ability of cyber-attack.

- `cyberAttack(Team enemyTeam)` — allows the student attack all alive members of the enemy team. The enemy does not receive any EP nor KP.

`SEStudent` – The *Software Engineering* students have special ability of group working and support.

- `groupWork(Character enemy)` — allows the all alive members of the team to attack the enemy. The friends and enemy do not receive any EP nor KP.
- `groupDiscussion()` — allows the student to heal all alive members of the team, the amount of HP gained is a half of the student's defence points. The friends do not receive any EP nor KP.

The special abilities can only be used when the student's KP reaches its maximum. Make sure that these method throws some `Exception` when this condition is not satisfied.

BY THIS STAGE you should have the first subclass of `Character`: `Student` class, and the subclasses of `Student`: `AIStudent`, `CSStudent`, `CyberStudent`, `SEStudent`. Figure 2 shows the relationship between the different classes that we have at the moment.

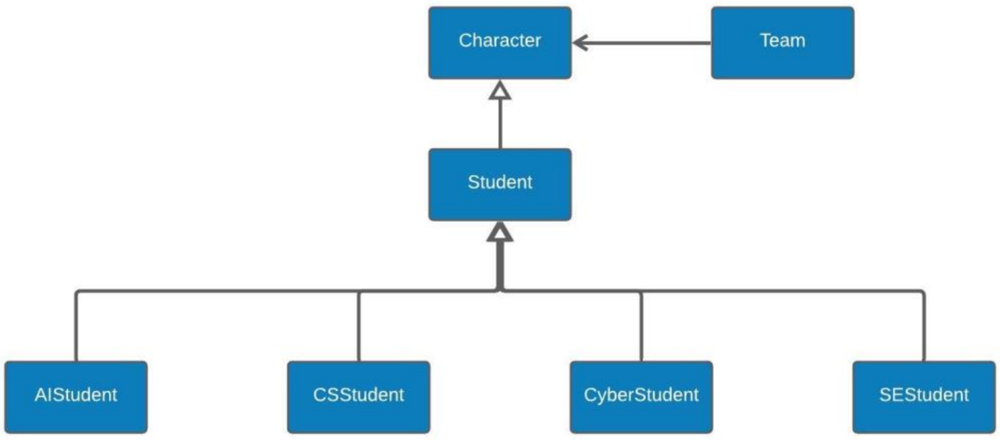
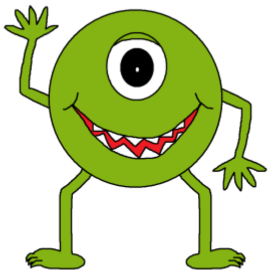


Figure 2. Different types of Students

You can now test your new Student class and its subclasses by creating a main method and to create some AIStudent, CSStudent, CyberStudent, SEStudent objects. You can check the students' abilities including their special abilities.

Part 3 – The Monster classes



Monsters

To model the various types of *Monsters*, we define an interface `Monster` which specifies the ability of the monsters. This interface must contain a method named `strike` for the monster to attack another character. Define the method signature as appropriate.

A monster has different abilities and they are invoked according to some probabilities depending on whether the monster is a M*inion* or a *Boss*. The list of monster's abilities are as follows.

- *SyntaxError* – the most common attack of the monster and that damages the enemy depending on the following formula

  100 * monster's attack attribute / (100 + enemy's defence attribute)
- *NullPointerException* – one of the most hated attacks of the monsters. This attack heals the monster by the amount the same as the monster's defence.
- *ArrayIndexOutOfBoundException* – another common attack of the monsters and deals double damage to the enemy
- *NoneTermination* – This attack is only available to a *Boss* and will resurrect all fallen members of the team.
- *ConcurrentModificationException* -- deals damages to all alive enemies. The damage to each enemy is calculated according to the formula above. This attack is only available to a *Boss*

For each attack, the monster's EP increases by 3. For each enemy that got attacked, their EP is increased by 3. If the enemy is a student, their KP is increased by 3. For each enemy killed, the monster gains an additional 4 EPs.

Minions and Bosses

The diagram in Figure 3 shows you how the `Minion` and `Boss` classes related to the other components: it inherits from the `Character` class and implements the `Monster` interface. Note that the diagram shows the minimum inheritance hierarchy for the classes, you can add more classes and additional inheritance structure as you like, as long as `Minion` and `Boss` classes are both `Character` and `Monster`.

Create constructors `Minion(String name)` and `Boss(String name)` that to construct the minions and bosses with the base attributes as showed in Table 1.

The implementation of the `strike` for the `Minion` class according to the following probabilities (75% *SyntaxError*, 15% *NullPointerException*, and 10% *ArrayIndexOutOfBoundException*).

The implementation of the `strike` for the `Boss` class according to the following probabilities (50% *SyntaxError*, 15% *NullPointerException*, 15% *ArrayIndexOutOfBoundException*, 10% *NoneTermination*, and 10% *ConcurrentModificationException*).

You can implement the probabilities with the Toolbox class that was provided with the labs, or by using the methods in the java.util.Random class directly.

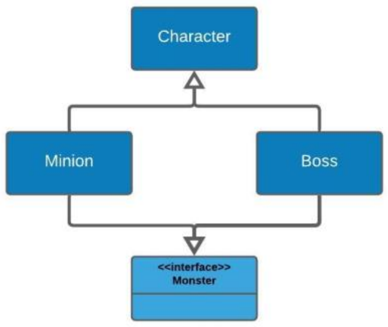


Figure 3. Inheritance Hierarchy of Monsters

BY THIS STAGE you should have the first interface `Monster` and the subclasses of `Character`: `Minion` and `Boss` implementing the `Monster` interface.

You can now test your new `Minion` and `Boss` classes by creating a main method and to create some monster objects, and check the monsters' abilities.

Part 4 – The Student Teams, Monster Teams, and Guilds

Student Teams and Monster Teams

The battle arena will be between teams of students and teams of monsters. Follow the instruction below to create the classes for `StudentTeam` and `MonsterTeam` accordingly.

Change the `Team` class to an abstract class and add an abstract method `move(Character member, Team enemyTeam)`. Parameter `member` indicates the member of the team who has the turn to move. Parameter `enemyTeam` specifies the opponent team. The method should throw an `Exception` if an illegal move for the character is performed.

Create two subclasses `StudentTeam` and `MonsterTeam` of `Team`. Each class must have a constructor with a `String` parameter representing the name of the team. Implement the method `move` accordingly for the two new classes. Ensure that you will not make any illegal move, e.g., that a student invokes a special ability when their KP is not yet reach its maximum.

Guilds

The teams are formed by members of a guild. Create a `Guild` class with a constructor taking no parameters, i.e., of the form `Guild()`. The `Guild` class must have the following methods.

- `addMember` – adds a character to the guild.
- `getMembers` – returns the members of the guild as a `Collection<Character>`.
- `getTeam` – given an enemy team, returns the team formed by the members of the guild. The maximum number of chosen members is 5.

BY THIS STAGE you should have implemented `StudentTeam` and `MonsterTeam` classes as subclasses of `Team`, and the Guild class. The relationships amongst these classes can be seen in Figure 4.

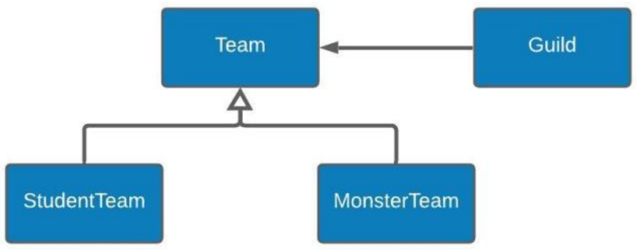


Figure 4. Inheritance Hierarchy of Monsters

You can now test your newly created classes by creating a main method and to create some teams and guilds. Call the `move` method on the teams and `getTeam` for guilds. You are advised to get all of this working before attempting to extend your functionality.

Part 5 – Battle

Our final simulation will contain several battles. We focus on simulating one battle in this part. You should create a new class called `Battle`. It should have two `Team` instances and have a `fight()` method which is used to run the simulation.

A typical battle contains several rounds until one of the team falls. The `fight()` method returns the winning team. To avoid a battle that can go too long, the maximum number of rounds for a battle is `30`. In the case where the maximum number of round has reached and both teams are survived, the battle is considered to be a draw and in this case, the `fight()` method returns `null`.

For each round of a battle, the following events occur.

- Sorting the alive members of both teams in the descending order according to their speeds. This will be the order for the member to move in the round.
- When a character has the turn, if the character is still alive, call the method on the character's team for the character to attack the enemy team, i.e., `move(Character member, Team enemyTeam)` of the `Team` class.

After each round it should call report on the teams to list information about members (including their current status).

You can use print statements to record what is happening in your simulation, including the characters moves. To slow things down a little, the following code will help you to pause for half a second between rounds if you choose to include it.

```
try
{
      Thread.sleep(500);
      // Code for a round
      ...
}
catch (InterruptedException e)
{
}
```

BY THIS STAGE you should have implemented `Battle` class, connected to two `Teams`

You can now test your new `Battle` class by creating a main method and to an instance of that for two Teams and call the corresponding `fight()` methods.

Part 6 – Tower of Monsters – Reading a simulation configuration file

In this part we implement a game where a guild of students combatting against several teams of monsters (tower of monsters). You will need to use your file handling methods as well as split strings into different component parts. The guild of students starts with 8 students, 2 from each programme, all starts at level 1.

Our basic configuration file will look like the example below. You may choose to extend this for your extensions, but for testing purposes your code should accept and use configuration files in this form. Each line gives information about a team of monsters.

```
Ardyn(Minion,1);Bahamut(Minion,1);Gilgamesh(Minion,1);Necron(Minion,1);Ultimecia(Minion,1)
Ardyn(Minion,2);Bahamut(Minion,1);Gilgamesh(Minion,1);Necron(Minion,1);Ultimecia(Minion,1)
Ardyn(Minion,2);Bahamut(Minion,2);Gilgamesh(Minion,1);Necron(Minion,1);Ultimecia(Minion,1)
Ardyn(Minion,2);Bahamut(Minion,2);Gilgamesh(Minion,2);Necron(Minion,1);Ultimecia(Minion,1)
Ardyn(Minion,2);Bahamut(Minion,2);Gilgamesh(Minion,2);Necron(Minion,2);Ultimecia(Minion,1)
Ardyn(Minion,2);Bahamut(Minion,2);Gilgamesh(Minion,2);Necron(Minion,2);Ultimecia(Boss,1)
Ardyn(Minion,2);Bahamut(Minion,2);Gilgamesh(Minion,2);Necron(Minion,2);Ultimecia(Minion,2)
Ardyn(Minion,3);Bahamut(Minion,2);Gilgamesh(Minion,2);Necron(Minion,2);Ultimecia(Minion,2)
Ardyn(Minion,3);Bahamut(Minion,3);Gilgamesh(Minion,2);Necron(Minion,2);Ultimecia(Minion,2)
Ardyn(Minion,3);Bahamut(Minion,3);Gilgamesh(Minion,3);Necron(Minion,2);Ultimecia(Minion,2)
Ardyn(Minion,3);Bahamut(Minion,3);Gilgamesh(Minion,3);Necron(Minion,3);Ultimecia(Minion,2)
Ardyn(Minion,3);Bahamut(Minion,3);Gilgamesh(Minion,3);Necron(Minion,3);Ultimecia(Boss,2)
```

Each line are list of monster separated by "`;`". Each monster has the following format.

```
Name(type, level)
```

Some example simulation files of varying complexity will be placed on the WIKI.

Create a class `TowerOfMonsters` with the `main()` method so that it can take a file on the command line. This will enable you to start your simulation by taking the name of the configuration file for the simulation.

```
java TowerOfMonsters monsters.txt
```

When the `TowerOfMonsters` receives the configuration, it should read the file a line at a time. For each line the *file* will need to create a team of monster corresponding to the line. You may find you need to create specific methods or indeed a helper class, to parse the configuration file and extract the information that the `TowerOfMonsters` needs.

For each level, choose a team of students from the guild to fight the team of monsters at that level. Create a battle between the teams. If the student team wins, move to the next level. If the student team loses, choose another team from the guild to fight again. If there are no students left in the guild, the game finishes.

You should now have a simulator that runs with a `TowerOfMonsters` containing a `Guild` which has 8 `Students` in it initially.

Exceptions

You should be trying to use exceptions in the construction of your simulator where possible. You should be catching appropriate I/O exceptions but also might consider the use of exceptions to correctly manage:

- The input of a configuration file that does not conform to the specified file format.
- …

Extensions

Before attempting any extension, please make sure you archive the basic part to a zip file for submission separately from the extensions.

You are free to extend your code beyond the basic simulator as described above. You are advised that your extensions should not alter the basic structures and methods described above as marking will be looking to see that you have met the specification as we have described it. Please include your extended version in a separate zip file.

Some extensions that we would heartily recommend include:

- Extend to include students from other programmes in ECS (from ELEC side), including their abilities.
- You might want to allow the simulation to save out the current state of a simulation to a file so that it can be reloaded and restarted at another time.

15% of the marks are available for implementing an extension. Any of the above examples would be suitable, but feel free to come up with one of your own if you like. It is not necessary to attempt multiple extensions in order to gain these marks. However, any extension that you proposed must be comparable in to the above examples in order to gain all the marks. Please describe your extension clearly in the readme file that you submit with your code.

Space Cadets

You might want to add a GUI to your simulator so that you can visualise the state of the *School* at any given moment. **No marks are available for a GUI**, we put the suggestion forward simply for the challenge of it, although you can reasonably expect praise and glory for your efforts.

Submission

Submit **all Java source files** you have written for this coursework in a zip file **`arena.zip`**. Do not submit class files. As a minimum this zip will include (at the top level without any folder):

`AIStudent.java`, `Battle.java`, `Boss.java`, `Character.java`, `CSStudent.java`, `CyberStudent.java`, `Guild.java`, `Minion.java`, `Monster.java`, `MonsterTeam.java`, `SEStudent.java`, `Student.java`, `StudentTeam.java`, `Team.java`, `TowerOfMonsters.java`

Please submit **all Java source files** you have written for extension in a zip file **`extension.zip.`** Do not submit class files.

Also include a text file **`readme.txt`** that contains a brief listing of the parts of the coursework you have attempted, describes how to run your code including command line parameters, and finally a description of the extensions you have attempted if any. Submit your files by **Friday 11th December 2020 16:00** to: https://handin.ecs.soton.ac.uk

Relevant Learning Outcomes

1. Simple object-oriented terminology, including classes, objects, inheritance and methods.
2. Basic programming constructs including sequence, selection and iteration, the use of identifiers, variables and expressions, and a range of data types.
3. Good programming style
4. Analyse a problem in a systematic manner and model in an object-oriented approach

Marking Scheme

Criterion	Description	Outcomes	Total
Compilation	Applications (containing all parts) that compile and run.	1	20

Specification	Meeting the specification properly.	1,2,4	50
Extensions	Completion of one or more extensions.	1,2,4	15
Style	Good coding style.	3	15