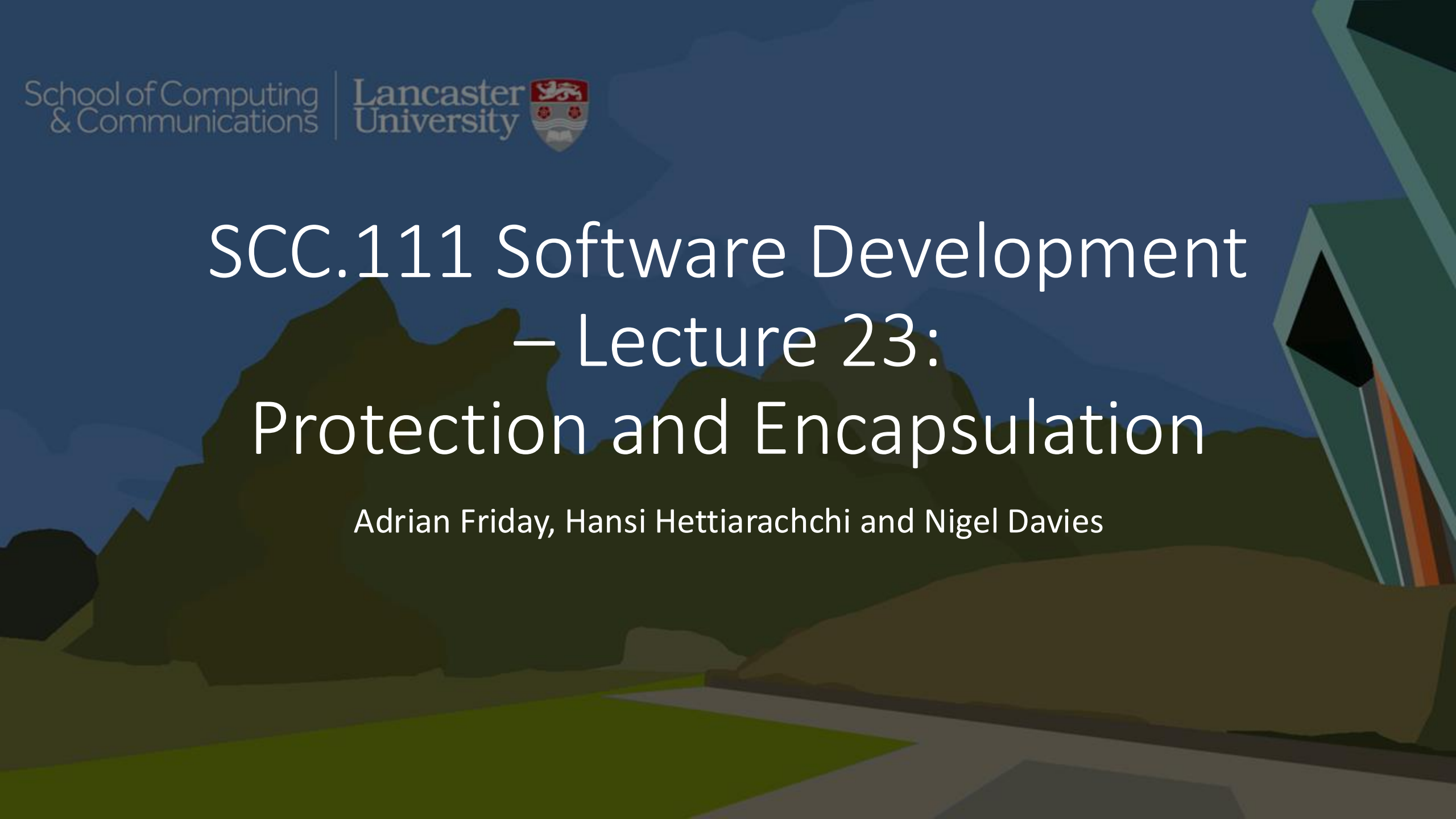

SCC.111 Software Development – Lecture 23: Protection and Encapsulation

Adrian Friday, Hansi Hettiarachchi and Nigel Davies

Last lecture…

Lancaster University

- Object Oriented programming is all about modularity
  - **Data + Methods = Object**
- Classes provide a specification of objects.
- Classes are **types.**
- Object instances are real, concrete realizations of classes.
- Using classes and instances
  - Use **class** to define classes
  - Use **the class name** to build object instances
  - Use **.** to access methods and attributes in an instance

```
class Car{
    int milesDriven = 0;

public:
    void drive(int miles);
};

void Car::drive(int miles){
    milesDriven = milesDriven + miles;
}

int main(){
    Car joesPassat;

    joesPassat.drive(16);
}
```

Our example class is not truly modular…

Lancaster University

- We can identify two key weaknesses with our class example:
  - **The class can be created without being initialised**
    - This leaves too much responsibility with the application programmer
    - Objects can be created in a useless state
  - **The attributes in the class are uncontrolled**
    - An application programmer has full access to variables and methods that are internal to the operation of a class
    - This is not consistent with the vision of a module
- These weaknesses relate to **protection**
  - Not all code is about doing stuff
  - Some code is about **preventing** stuff from happening as a duty of care

Modularity

Lancaster University

Modularity relies on **encapsulation**

The module must contain its inner workings…

…protect them…

… and only expose functionality through its interfaces.

Access control: private and public

Lancaster University

- Attributes and methods can be declared **public**
  - These **can be accessed from inside or outside the class.**
  - Use this to present an Application Programmable Interface (API).

- Attributes and methods can be declared **private**
  - These can only be accessed from methods **inside** the class.
  - They **cannot be accessed from methods outside the class.**
  - Use this to ensure your class remains modular, without exposing more functionality than you want to outside the class.

- **Always consider which attributes and methods should public or private.**
- **As this defines the API of the code you write.**

Lancaster University

Encapsulation in C++

Class

```
class Car{
private:
    int milesDriven = 0;
    char *colour;
public:
    void drive(int miles);
    void respray(char *c);
    void show();
};
```

Access control: accessors and mutators

Lancaster University

- **It is vital to control class attributes**
  - If a programmer has access, they can directly read or write that attribute.

- It is rare that as a developer of a class you want other programmers to have the power to come into your code and change your attributes!

- As a point of good practice, you should never create public class attributes in OO languages. Make the attribute private and use accessor/mutator methods as appropriate*.

Access control: accessors (get methods)

Lancaster University

- **public** methods that **wrap** access to a **private** attribute.
  - They are named after the variable they relate to.
  - They are declared public.
  - The have no parameters.
  - They return the value of the variable they relate to.

```
class Car {
    private:
        char *colour;

    public:
        char *getColour();
};

char *Car::getColour() {
        return colour;
    }
```

Access control: mutators (set methods)

Lancaster University

- **public** methods that **change** the value of a **private** instance variable
  - They are named after the variable they relate to.
  - They are declared public.
  - They have a single parameter, same type as the instance variable.
  - They set the value of the variable to the given parameter.
  - They have no return value (void).

```
class Car {
    private:
        char *colour;

    public:
        void setColour(char *c);
};

void Car::setColour(char *c) {
        colour = c;
}
```

Constructors 1

Lancaster University

Constructors help you to initialize your objects

- A constructor is a method that is called automatically when an object instance of that class is created (i.e when someone creates a new object).
- Provides a way for you to initialize the values of your object's instance variables to a consistent state.
- All classes have a constructor. If you don't define one, the compiler will create an empty one for you.

Constructors look like all other methods, but with no return type

Constructors 2

Lancaster University

```
class Car {
    public:
        Car();
};

Car::Car(){
    // Initialisation code goes here
}
```

```
int main(){
    Car newCar; // the constructor is called here
}
```

Constructors 3

Lancaster University

Constructors can have any number of parameters (like any method)…

- These are then **required** when an instance is created
- A mechanism to ensure that the programmer using your class provides you with the information you need to build an instance in its initial state

```
class Car {
    private:
        char *colour, *brand;
        int milesDriven;

    public:
        Car(char *col, char *b);
}
Car::Car(char *col, char *b) {
    colour = col;
    brand = b;
    milesDriven = 0;
}
```

```
int main() {

   Car joesCar((char *)"White", (char *)"VW");
        //this is correct
   Car saadsCar((char *)"Grey", (char *)"Benz");
        //this is correct
   Car anotherCar();
        // this is incorrect!

   return 0;
}
```

Constructors 4

Lancaster University

A class can have many constructors…

- Any can be used when creating an object instance of your class
- They must all have unique parameter list
- The parameter list is used to identify which constructor to call

```
int main() {
    Car newCar((char *)"White", (char *)"VW");
        // calls 1st constructor
    Car oldCar((char *)"Grey", (char *)"Benz",
        120000);
        // calls 2nd constructor
    Car anotherCar; // this is incorrect!

    return 0;
}
```

```
class Car {
    private:
        char *colour, *brand;
        int milesDriven;

    public:
        Car(char* col, char* br);
        Car(char* col, char* br, int m);
};

Car::Car(char* col, char* br) {
    colour = col;
    brand = br;
    milesDriven = 0;
}
Car::Car(char* col, char* br, int m) {
    colour = col;
    brand = br;
    milesDriven = m;
}
```

Why?

Lancaster University

Safety

- Programmers **cannot** create invalid/inconsistent variable values.
- Either by accident, or maliciously.
- This promotes simpler, safer code…

Control

- The programmer writing a class can now define and enforce its API

```
class BankAccount{
 private:
    int balance = 0;

 public:
    void withdraw(int);
    void deposit(int);
};

void BankAccount :: withdraw(int amount){
    if (balance >= amount){
        balance -= amount;
        printf("%d GBP was successfully withdrawn\n",
        amount);
   } else {
        printf("Insufficient balance\n");
   }
}
```

Destructors

Lancaster University

- **A destructor is a method that is invoked automatically whenever an object is going to be destroyed.**

- This occurs when:
  - the variable relating to that object goes out of scope.
  - the object is explicitly deleted.

```
class BankAccount {

    // Attributes and methods

    // Destructor
    ~BankAccount();
};

BankAccount::~BankAccount()
{
    // bank collects remaining
    // balance… ☺
}
```

Summary

Lancaster University

- Today we learned :
  - Why we need to protect our classes from misuse
  - How to use public and private to control visibility of our code to other programmers
  - How constructors ensure objects are created in a known state
  - How to implement these principles in C++