First, a Quick Review!

What do we mean by a *Class*?

Classes may be _inherited_

Classes may be _abstract_

But what does this mean?

UNIVERSITY OF TORONTO MISSISSAUGA

Classes, attributes and methods may be

public

private

or ….

Classes, attributes and methods may be

static

non-static

UNIVERSITY OF TORONTO MISSISSAUGA

How many Classes can any one Class *inherit?*

How many interfaces can any one Class *implement?*

UNIVERSITY OF TORONTO MISSISSAUGA

What do we mean by *polymorphism*?

How does this relate to *classes*, or *interfaces*?

UNIVERSITY OF TORONTO MISSISSAUGA

Object Oriented Design Principles

Classes and interfaces provide a level of **abstraction** that hide internal implementation details. Clients need to know the methods available to call and the parameters that are required, but not the implementation details.

Data and methods are **encapsulated** within each Class, i.e. bundled together and hidden.

We bundle methods and data to promote **Low Coupling** of classes, meaning that changes to one class should not necessitate changes to many others.

We bundle methods and data to promote **High Cohesion,** meaning to co-locate methods with strongly-related functionality.

Object Oriented Design Principles: SOLID

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

From "Design Principles and Design Patterns," Robert C. Martin, 2000

Object Oriented Design Principles: SOLID

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

We'll be seeing these principles come up many times this term, so get comfortable with them now!

Single Responsibility Principle

Every class should have a **single responsibility.**

Every class should have **one reason to change.**

“When you write a software module, you want to make sure that when changes are requested, those changes can only originate from … a single tightly coupled group of people representing a single narrowly defined business function. You want to isolate your modules from the complexities of the organization as a whole, and design your systems such that each module is responsible (responds to) the needs of just that one business function.”

Robert C. Martin

Single Responsibility Principle

```java
public class Invoice {
    5 usages
    private Book book;
    3 usages
    private int quantity;
    3 usages
    private double discountRate, taxRate, total;

    public Invoice(Book book, int quantity, double discountRate, double taxRate) {
        this.book = book;
        this.quantity = quantity;
        this.discountRate = discountRate;
        this.taxRate = taxRate;
        this.total = this.calculateTotal();
    }

    1 usage
    public double calculateTotal() {
        return ((book.price - book.price * discountRate) * this.quantity) * (1 + taxRate);
    }

    public void printInvoice() {
        System.out.println(quantity + "x " + book.name + " " + book.price + "$");
        System.out.println("Discount Rate: " + discountRate);
        System.out.println("Tax Rate: " + taxRate);
        System.out.println("Total: " + total);
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }
}
```

What might be a problem here? What pieces might change?

Example from [Yiğit Kemal Erinç](), The SOLID Principles of Object-Oriented Programming Explained in Plain English

Open/Closed Principle

Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.

Add new features **not by modifying** the original class, but rather by **extending** it.

Liskov Substitution Principle

If S is a subtype of (i.e. inherited or derived from) T, then objects of type S may be substituted for objects of type T without altering any of the desired properties of the program.

For example, if S is a **child class** of T, then we should be able to **substitute** T for S wherever it appears without breaking the code.

Barbara H. Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16, 6 (Nov. 1994), 1811–1841. https://doi.org/10.1145/197320.197383

Liskov Substitution Principle

```java
class Rectangle {
    protected int width, height;

    public Rectangle() {
    }

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}
```

```java
class Square extends Rectangle {
    public Square() {}

    public Square(int size) {
        width = height = size;
    }

    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

Liskov Substitution Principle

```
static void getAreaTest(Rectangle r) {
    int width = r.getWidth();
    r.setHeight(10);
    System.out.println("Expected area of " + (width * 10) + ", got " + r.getArea());
}

public static void main(String[] args) {
    Rectangle rc = new Rectangle(2, 3);
    getAreaTest(rc);

    Rectangle sq = new Square();
    sq.setWidth(5);
    getAreaTest(sq);
}
```

What is the problem here? How might you rewrite the class?

Example from [Yiğit Kemal Erinç](), The SOLID Principles of Object-Oriented Programming Explained in Plain English

Interface Segregation Principle

The use of the word *interface* here means all of the public methods associated with a class. Every instance of the class must have a valid implementation of these public methods, as clients of the class may depend on them.

But no one should be forced to implement irrelevant methods in an interface!

It is therefore better to have lots of **small, specific interfaces** than a few larger ones; this makes the software **easier to extend and modify.**

Interface Segregation Principle

```java
public class FreeParking implements ParkingLot {

    @Override
    public void parkCar() {

    }

    @Override
    public void unparkCar() {

    }

    @Override
    public void getCapacity() {

    }

    @Override
    public double calculateFee(Car car) {
        return 0;
    }

    @Override
    public void doPayment(Car car) {
        throw new Exception("Parking lot is free");
    }
}
```

```java
public interface ParkingLot {

    void parkCar(); // Decrease empty spot count by 1
    void unparkCar(); // Increase empty spots by 1
    void getCapacity(); // Returns car capacity
    double calculateFee(Car car); // Returns the price based on number of hours
    void doPayment(Car car);
}
```

What is the problem here? How might you rewrite the interface?

Example from [Yiğit Kemal Erinç](), The SOLID Principles of Object-Oriented Programming Explained in Plain English

Dependency Inversion Principle

Reduce chains of dependence between classes so that you can change an individual piece without having to change anything more than the individual piece.

There are two aspects to the dependency inversion principle:

High-level modules should not depend on low-level modules. Both should depend on **abstractions**.

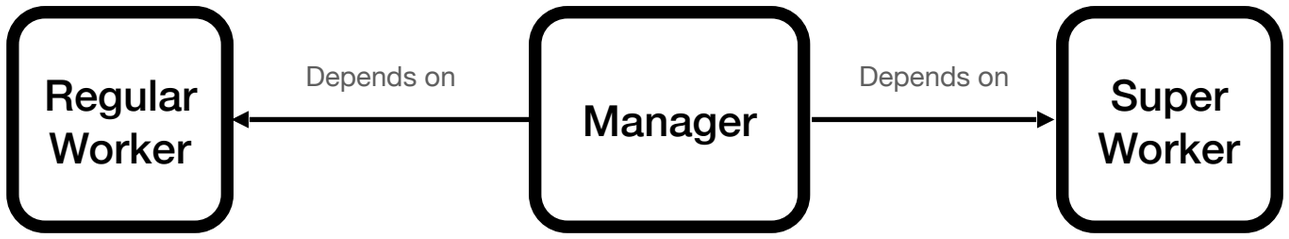
Abstractions should not depend upon concrete details. Details should depend upon **abstractions**.

Dependency Inversion Principle

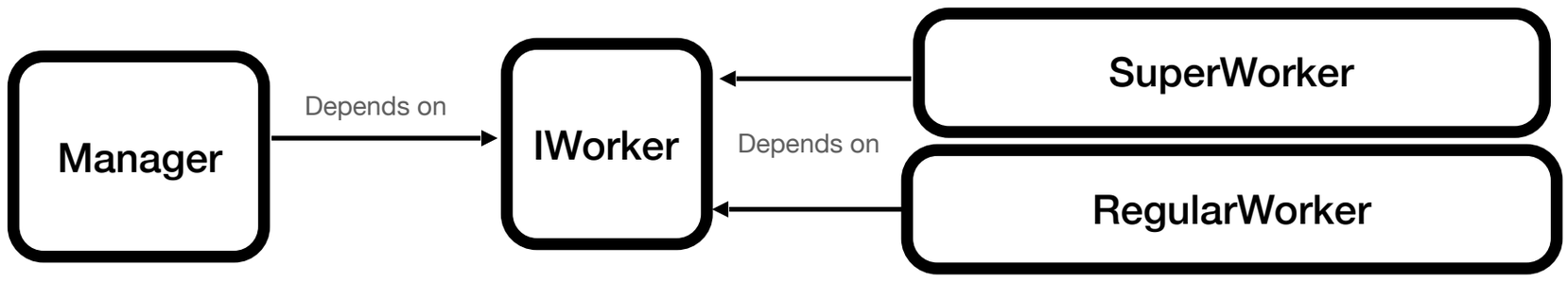
Say you have a Manager and a RegularWorker Class, and the Manager class has several methods that have RegularWorker parameters.

Someone now adds a SuperWorker class. How much of our code needs to be rewritten?

We can solve this kind of problem by removing the dependency between Manager and Worker.



We can instead design an interface (IWorker) for all worker types to use. Everyone now depends on the IWorker interface, but there are no direct dependencies between classes.



SOLID Principle Question #1

Imagine you're writing software to keep track of cars. You've designed an interface called Car with the following two methods:

turnOnEngine();

accelerate();

Your employer asks you to modify your code to keep track of an electric car. But electric cars have no engine!

Should ElectricCar implement the Car interface? Why or why not?

What SOLID Principles apply to this problem?

UNIVERSITY OF TORONTO MISSISSAUGA

SOLID Principle Question #2

Imagine you are refactoring software to keep track of computers. You come across a class called WindowsBox which looks like this:

```
public class WindowsBox {

    private final Keyboard keyboard;
    private final Monitor monitor;

    public WindowsBox() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }
}
```

This works, but it creates a dependence between the WindowsBox class and the Keyboard and Monitor class.

How might we rewrite this to promote less dependency between concrete classes and more abstraction?

What SOLID Principles apply to this problem?

Example from https://www.baeldung.com/solid-principles

SOLID Principle Question #3

Imagine you are refactoring software to keep track of cat owners. You come across an interface called KeepCat which looks like this:

```
public interface KeepCat {
    void changeCatLitter();
    void feedCat();
    void petCat();
    void brushCat();
}
```

Some cat owners co-parent cats, however. They may only change the litter for their cat, while their partner handles the brushing!

How might we rewrite this to promote more flexibility in our assignment of behaviours to cat owners?

What SOLID Principles apply to this problem?

UNIVERSITY OF TORONTO MISSISSAUGA