LECTURE 1

Programming Paradigms

- **Imperative (procedural)**
  - Specifies *how* computation proceeds using statements that change program state
- **Declarative**
  - Specifies *what* should be computed, rather than how to compute it
- **Functional**
  - A form of declarative programming and treats computation like *evaluating mathematical functions*
- **Object-oriented**
  - Supports imperative programming but *organizes programs as interacting objects*, following the real-world

Primitive data-type: numerical, character, boolean
Composite data-type:

- Homogeneous: array (multi-dimensional)
- Heterogeneous: record (or structure)

Static Typing vs Dynamic Typing

- Dynamic (e.g. JavaScript):

```
var a;
var b = 5.0;
var c = "Hello";

b = "This?"; // ok
```

- Static (e.g. Java):

```
int a;
double b = 5.0;
String c = "Hello";

b = "This?"; // error
```

- Need to develop a sense of "type awareness" by maintaining type-consistency
- During compilation, incompatible typing throws off a compile-time error

Modularity

- Taking a complex program and breaking it up into dedicated sub-tasks to be solved
- The `main` method (object-oriented equivalent of function/procedure) describes the solution in terms of higher-level *abstractions*

```
import java.util.Scanner;

class DiscCoverage {

    public static void main(String[] args) {
        double[][] points;

        points = readPoints();
        printPoints(points);
    }
```

- Abstractions can then be solved *individually* and *incrementally*

OOP Principle #1: Abstraction

- Establish an abstraction level relevant to the task at hand and ignore lower level details
  - Data abstraction: abstract away low level data items
  - Functional abstraction: abstract away control flow details

OOP Principle #2: Encapsulation

- **Abstraction barrier**
  - Separation between implementer and client
- Having established a particular high-level abstraction,
  - Implementer **defines** the data/functional abstractions using lower-level data items and control flow
  - Client **uses** the high-level data-type and methods
- **Encapsulation**
  - To protect implementation against inadvertent use
  - Packaging data and related behaviour together into a self-contained unit
  - Hiding information/data from the client, restricting access using methods/interfaces

LECTURE 2

Overriding toString method

```
@Override
public String toString() {
    return "Circle with area " +
        String.format("%.2f", getArea()) + " and perimeter " +
        String.format("%.2f", getPerimeter());
}
```

The annotation `@Override` indicates to the compiler that the method overrides another one

Overriding Object's equals Method

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    } else if (obj instanceof Circle) {
        return this.radius ==
            ((Circle) obj).radius;
    } else {
        return false;
    }
}
```

Inheritance

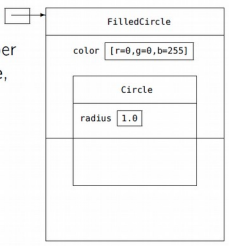
- Notice the child class `FilledCircle` invokes the parent class `Circle`'s constructor using `super(radius)` within it's own constructor
- The `radius` variable in `Circle` can also be made accessible to the child class by changing the access modifier

```
public class Circle {
    protected double radius;
    ...
```

- The `super` keyword is used for the following purposes:
  - `super(..)` to access the parent's constructor
  - `super.radius` or `super.getArea()` can be used to make reference to the parent's properties or methods; especially useful when there is a conflicting property of the same name in the child class

Modeling Inheritance

- Notice how the child object "wraps-around" the parent
- Type-casting a child object to a super class, e.g. `(Circle) filledCircle`, refers to the parent object where attributes/methods can be assessed
- The only exception is overridden methods; calling them through the parent or child will invoke the overriding methods
- An overridden parent method can only be called within the child class via `super`



Polymorphism

- How is inheritance useful?
- Other than as an "aggregator" of common code fragments in similar classes, inheritance is used to support **polymorphism**
- Polymorphism means "many forms"

Static binding

- Given an array `Circle[] circles` comprising both `Circle` and `FilledCircle` objects, output these objects one at a time
- In static (or early) binding, we can do something like this:

```
for (Circle circle : circles) {
    if (circle instanceof Circle) {
        System.out.println((Circle) circle);
    } else if (circle instanceof FilledCircle) {
        System.out.println((FilledCircle) circle);
    }
}
```

- Static binding occurs during compile time, i.e. all information needed to call a specific method can be known at compile time

Method Overloading

- Static binding also occurs during method overloading
- Method overloading commonly occurs in constructors

```
public Circle() {
    this.radius = 1.0;
}

public Circle (double radius) {
    this.radius = radius;
}
```

- Whichever method is called is determined during compile time

```
Circle c1 = new Circle();
Circle c2 = new Circle(1.2);
```

- Methods of the same name can co-exist if the *signatures (number, order, and type of arguments)* are different

Dynamic binding

- Contrast static binding with dynamic (or late) binding

```
for (Circle circle : circles) {
    System.out.println(circle);
}
```

- The above will give the same output as in the previous case
- Notice that the exact type of circle, and the exact `toString` method to be overridden, is not known until runtime

Class Variables and Methods

Use the `static` modifier to create class variables and methods

```
public class Circle {
    private double radius;
    private static int numOfCircles = 0;

    public Circle(double radius) {
        this.radius = radius;
        numOfCircles++;
    }

    public static int getNumOfCircles() {
        return numOfCircles;
    }
```

- Class variables and methods can be called through the class or the object
- Calling through the class is preferred as it makes clear the intent

LECTURE 3

Abstract Classes

- Redefine `Shape` as an **abstract** class with abstract methods; these methods are to be implemented in the child classes

```
public abstract class Shape {

    public abstract double getArea();

    public abstract double getPerimeter();

    @Override
    public String toString() {
        return "Area " +  getArea() +
            " and perimeter " + getPerimeter();
    }
}
```

Shapes, Circles and Rectangles Revisited

- An alternative design for constructing shape objects (i.e. circles and rectangles) is to decide on what common **behaviour** each shape object should provide
- In our example, each shape
  - can return an area
  - can return an perimeter
  - can return a string representation for output purposes
- The above defines a `Shape` "contract" between what the user expects of the implementer of a shape object
- In Java, the contract takes the form of an **interface**

- The `Shape` interface is defined as

```
public interface Shape {
    static final double PI = 22.0 / 7;

    public double getArea();

    public double getPerimeter();

    @Override
    public String toString();
}
```

- Note that an interface does not allow for instance properties; apart from constant declarations (e.g `PI` defined above)
- Just like an abstract class, interfaces cannot be instantiated

Interfaces support Polymorphism

- inheritance (via **`extends`**) depicts an **is–a** relationship
- On ther other hand, interfaces (via **`implements`**) depicts a
  - **can–do** relationship
  - **is–a** relationship towards a non-concrete super-class
- Both interface and inheritance allows a `Circle` (`Rectangle`) to take on the form of a `Shape` — polymorphism

Polymorphic `Shape` Objects

- Notice how polymorphism, as well as dynamic (or late) binding, takes effect in the same way as inheritance

```
class Main {
    public static void main(String[] args) {
        Shape[] shapes = {new Circle(1.0),
            new Rectangle(8.9, 1.2)};

        for (Shape shape : shapes) {
            System.out.println(shape);
        }
    }
}
```

Inheritance vs Interface

- While both inheritance and interfaces supports polymorphism, there is one important difference between them
- A class can only inherit from one parent class, but a class can implement many interfaces
  - Java prohibits multiple inheritance to avoid common-sense ambiguities, e.g. *spork is both a spoon and a fork*

Access Modifiers

- The access level (most restrictive first) is given as follows:
  - private (visible to the class only)
  - default (visible to the package)
  - protected (visible to the package and all sub classes)
  - public (visible to the world)

Creating Packages

- Let's use an example where we desire to have `Shape`, `Scalable`, `Circle` and `Rectangle` classes/interfaces reside in the `cs2030.shapes` package
- Include the following line at the top of the java files

```
package cs2030.shapes;
```

- Compile the four Java files using

```
javac -d . *.java
```

- This will create the `cs2030/shapes` directory with the associated class files stored within

Preventing Inheritance and Overriding

- We have seen how the **`final`** keyword can be used to create final variables, or variables containing values that cannot be changed; in other words, constants
- The **`final`** keyword can also be applied to methods or classes
- Sometimes we need to prevent a class from being inherited
  - As an example, `java.lang.Math` and `java.lang.String` classes cannot be inherited from
  - We can use the **`final`** keyword to explicit prevently inheritance

```
public final class Circle {
    :
}
```

- We can also allow inheritance but prevent overriding

```
public class Circle {
    :
    @Override
    public final double getArea() {
        :
    }
    :
    @Override
    public final double getPerimeter() {
        :
    }
}
```

LECTURE 4

```
public static void main(String[] args) {

    FileReader file = new FileReader(args[0]);
    Scanner scanner = new Scanner(file);
    List<Point> points = new ArrayList<>();

    while (scanner.hasNextDouble()) {
        double x = Double.parseDouble(scanner.next());
        double y = Double.parseDouble(scanner.next());
        points.add(new Point(x, y));
    }

    DiscCoverage maxCoverage = new DiscCoverage(points);
    System.out.println(maxCoverage);
}
```

- Compiling the program gives the following compilation error:

```
Main1.java:12: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
        FileReader file = new FileReader(args[0]);
                          ^
1 error
```

- From the Java API Specifications for `FileReader` the following constructor is specified:

```
public FileReader(String fileName)
          throws FileNotFoundException
```

- This means that the `FileNotFoundException` must be handled (or thrown)
- One way is to just throw the exception out from the `main` method in order to make it compile

```
public static void main(String[] args) throws FileNotFoundException {
```

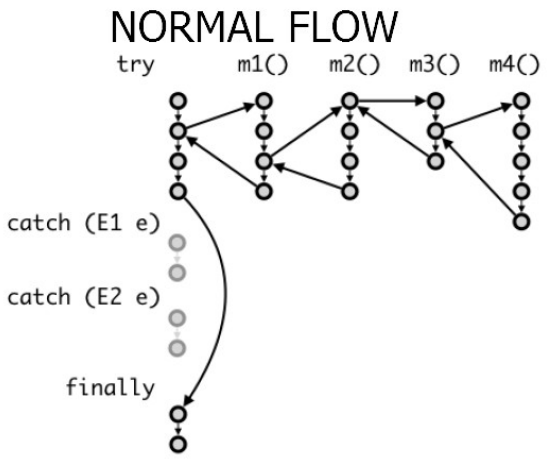
Handling Exceptions

- The more responsible way is to handle the exception:

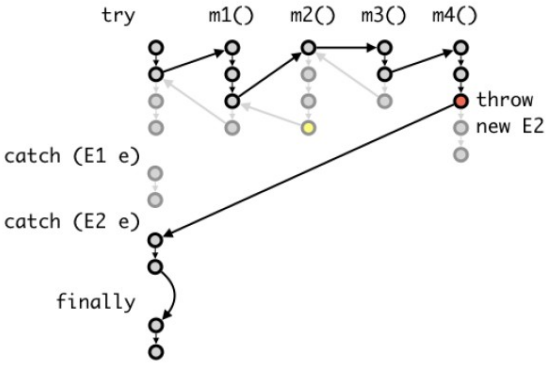
```
try {
    FileReader file = new FileReader(args[0]);
    Scanner scanner = new Scanner(file);
    List<Point> points = new ArrayList<>();

    while (scanner.hasNext()) {
        double x = Double.parseDouble(scanner.next());
        double y = Double.parseDouble(scanner.next());
        points.add(new Point(x, y));
    }

    DiscCoverage maxCoverage = new DiscCoverage(points);
    System.out.println(maxCoverage);

} catch (FileNotFoundException ex) {
    System.err.println("Unable to open file " + args[0] +
            "\n" + ex);
} catch (ArrayIndexOutOfBoundsException ex) {
    System.err.println("Missing filename");
} catch (NumberFormatException | NoSuchElementException ex) {
    System.err.println("Incorrect file format\n");
} finally {
    System.err.println("Program Terminates\n");
}
```

`try` and `catch` Blocks

- Notice that while error (exception) handling is performed, the business logic of the program does not change
- This is made possible because of separate **`try`** and **`catch`** blocks; specifically
  - The **`try`** block encompasses the business logic of the program
  - Exception handling is dealt with in separate **`catch`** blocks, typically one for each exception
  - In addition, there is an optional **`finally`** block which can be used for house-keeping tasks
- Exceptions provide us a way to keep track of the reason for program failure, without which we would then have to rely on error numbers stored in normal variables
- More than one exception can be handled in a single catch block using `|`
- An exception (just like an object) can be printed, typically through `System.err.println`



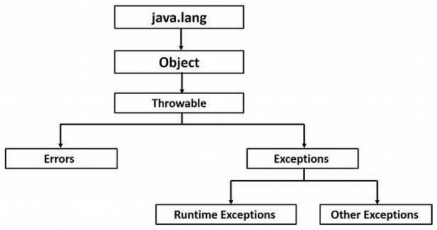
Suppose an exception `E2` is thrown in `m4()`, and causes the execution in `m4` to stop prematurely



Types of Exceptions

- There are two types of exceptions:
  - A **checked exception** is one that the programmer should actively anticipate and handle
  - An **unchecked exception** is one that is unanticipated, usually the result of a bug
- All checked exceptions should be caught (**`catch`**) or propagated (**`throw`**)

Exception Hierarchy



- Unchecked exceptions are sub-classes of `RuntimeException`
- All `Errors` are also unchecked.

Generating Exception

- Create your own exception by inheriting from existing ones

```
class IllegalCircleException extends IllegalArgumentException {
    Point p;
    Point q;

    IllegalCircleException(String message) {
        super(message);
    }

    IllegalCircleException(Point p, Point q, String message) {
        super(message);
        this.p = p;
        this.q = q;
    }

    @Override
    public String toString() {
        return p + ", " + q + ": " + getMessage();
    }
}
```

Notes on Exceptions

- When overriding a method that throws a checked exception, the overriding method must throw only the same or more specific exception (why?)

Assertions

- These conditions are called **assertions**; there are two types:
  - **Preconditions** are assertions about a program's state when a program is invoked
  - **Postconditions** are assertions about a program's state after a method finishes
- There are two forms of `assert` statement
  - `assert expression;`
  - `assert expression1 : expression2;`

example :

```
public double distanceTo(Point q) {
    double distance = Math.sqrt(Math.pow(dx(q),2) +
           Math.pow(dy(q),2));
    assert distance >= 0;
    return distance;
}
```

For a more meaningful message, replace the assertion with

```
assert distance >= 0 :
    this.toString() + " " + q.toString() + " = " + distance;
```

Enumeration

- An **enum** is a special type of class used for defining constants
- To define an **enum**,

```
enum EventType {
   ARRIVE,
   SERVE,
   WAIT,
   LEAVE,
   DONE
}
```

- **enum** are type-safe since `eventType = 1` no longer works, but `eventType = EventType.ARRIVE` does

Enum's Fields and Methods

- Each constant of an **enum** type is an instance of the **enum** class and is a field declared with **public static final**
- Constructors, methods, and fields can be defined in **enum**s

```
enum Color {
    BLACK(0, 0, 0),
    WHITE(1, 1, 1),
    RED(1, 0, 0),
    BLUE(0, 0, 1),
    GREEN(0, 1, 0),
    YELLOW(1, 1, 0),
    PURPLE(1, 0, 1);

    private final double r;
    private final double g;
    private final double b;

    Color(double r, double g, double b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public double luminance() {
        return (0.2126 * r) + (0.7152 * g) +
            (0.0722 * b);
    }

    public String toString() {
        return "(" + r + ", " + g + ", " + b + ")";
    }
}
```

LECTURE 5

Abstraction Principle

- "Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts" — *Benjamin C. Pierce*

Single Responsibility Principle (SRP)

- "A class should have only one reason to change." — *Robert C. Martin (aka Uncle Bob)*
- Responsibility is defined as the "reason to change"
- A class/module should do only one thing and do it well

Open-Closed Principle (OCP)

- Uncle Bob simply states it as

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

Liskov Substitution Principle (LSP)

- Introduced by Barbara Liskov

  > "Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T."

- This **substitutability** principle means that if S is a subclass of T, then an object of type T can be replaced by that of type S without changing the desirable property of the program.

Interface Segregation Principle (ISP)

- Uncle Bob (again)

  > "no client should be forced to depend on methods it does not use."

- Keep interfaces minimal; avoid "fat" interfaces
  - Do not force classes to implement methods that they can't
  - Do not force clients to know of methods in classes that they don't need to
- Split interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them

Programming to an Interface

- **Program to an interface, not an implementation**
- A client should always use the interfaces from an application rather than the concrete implementations
- The benefits of this approach are **maintainability**, **extensibility** and **testablility**

LECTURE 6

- Java API provides **collections** to store groups of related objects together
  - provides methods that organize, store and retrieve data
  - there is no need to know how data is being stored
- `ArrayList<E>`: type parameter `E` replaced with type argument to support **parameterized types**, e.g. `ArrayList<Circle>`

```
ArrayList<Circle> = new ArrayList<Circle>();
```

- **Generic classes**: classes that allow some type parameter
- Convention: `T` for type; `E` for element; `K` for key; `V` for value
- Diamond notation `<>` lets the compiler infer the element type from the declaration; the following is equivalent

```
ArrayList<Circle> numbers = new ArrayList<>();
```

Auto-boxing and Unboxing

- Only reference types allowed as type arguments; primitives need to be auto-boxed/unboxed, e.g. `ArrayList<Integer>`
- Placing an `int` value into `ArrayList<Integer>` causes it to be **auto-boxed**
- Getting an `Integer` object out of `ArrayList<Integer>` causes the `int` value inside to be **(auto-)unboxed**

From Sub–Class to Sub–Types

- Recall in LSP, if S is a sub-class of T, then object of type T can be replaced with that of type S without changing the desirable property of the program
- Moreover, S is a **sub-type** of T if a piece of code written for variables of type T can be safely used on variables of type S
- Let `S` and `T` represent classes or interfaces, and `S <: T` denote `S` being a sub-type of `T`
  - Is `S[] <: T[]`? YES
    e.g. `Shape[] shapes = new Circle[10];`
  - Is `S<E> <: T<E>`? YES
    e.g. `List<Point> points = new ArrayList<Point>();`
  - Is `C<S> <: C<T>`? NO
    e.g. `ArrayList<Shape> shapes = new ArrayList<Circle>();`

Wildcards

- How do we then sub-type among generic types, in the spirit of `Shape[] shapes = new Circle[10];`
- The answer is to use the wildcard `?` such as

```
ArrayList<?> anyList = new ArrayList<Circle>();
```

- Even though `?` seems analogous to type `Object`, the **wildcard is not a type**
  - cannot declare a class of parameterized type `?`
  - use when specifying type of variable, field or parameter

Upper-Bounded Wildcards

```
static void readBurger(List<? extends Burger> burgerProducer) {
    for (Burger burger : burgerProducer) {
        System.out.println(burger);
    }
}
```

- `? extends Burger` means any type that extends from `Burger`, including itself

Lower-Bounded Wildcards

```
static void addBurger(List<? super Burger> burgerConsumer) {
    burgerConsumer.add(new Burger());
}
```

- `? super Burger` means any type that `Burger` extends from (i.e. super-type of `Burger`), including itself

Generic Methods

- Consider the following:

```
Integer[] nums = {19, 28, 37};
System.out.println(max3(nums));
```

- Other than using `Integer` class, can define generic methods

```
public static <T extends Comparable<T>> T max3(T[] nums) {
    T max = nums[0];

    if (nums[1].compareTo(max) > 0) {
        max = nums[1];
    }

    if (nums[2].compareTo(max) > 0) {
        max = nums[2];
    }

    return max;
}
```

`List<E>` Interface

- `List<E>` interface also specifies a sort method

```
default void sort(Comparator<? super E> c)
```

- Interface with `default` method indicates that `List<E>` comes with a default sort implementation
  - A class that implements the interface need not implement it again, unless the class wants to override the method

Comparator

- `sort` method takes in an object `c` with a generic **functional interface** `Comparator<? super E>`
  - `compare(o1, o2)` should return `0` if the two elements are equals, a negative integer if `o1` is "less than" `o2`, and a positive integer otherwise

given `Burger <: FastFood`

- covariant: `Burger[] <: FastFood[]`
- invariant: `C<Burger>` and `C<FastFood>`
- covariant: `C<Burger> <: C <? extends FastFood>`
- contravariant: `C<FastFood> <: C<? super Burger>`

MISCELLANEOUS

17. Class modifier appears in the following order:

```
1  public protected private abstract default static final
```

Reading from file

```
FileReader file = new FileReader(args[0]);
Scanner scanner = new Scanner(file);
```

List Of important imports :

- Java.util.ArrayList
- Java.util.List
- Java.io.FileReader
- Java.util.Scanner