# CSCI 132:
# Basic Data Structures and Algorithms

Static methods, Polymorphism, Exceptions,
Abstract Classes, Debugging

Reese Pearsall

Spring 2024

Opening a project in Eclipse

**Static methods** are methods in Java that can be called without creating an object of a class

```
public class StaticDemo {
        public static void fun1(String arg1) {
                System.out.println(arg1);
        }
        public static void main(String[] args) {
                fun1("Hello");
        }
}
```

I do **not** need to create a `StaticDemo` object in order to call the `fun1()` method

**Static methods** are methods in Java that can be called without creating an object of a class

```java
public class StaticDemo {
        public static void main(String[] args) {
                AnotherClass.funMethod("Hello");
        }
}
```

StaticDemo.java

```java
public class AnotherClass {
    public static void funMethod(String arg)
    {
        System.out.println(arg);
    }
}
```

AnotherClass.java

If the static method is in another class, we can access it by giving the class name (`AnotherClass`)

Once again, I do not need to create an `AnotherClass` object to call this static method

However, now objects are no longer an implicit argument to this method (cant use `this` anymore)

**Static methods** are methods in Java that can be called without creating an object of a class

```
Error: static method cannot be referenced from a non static context
```

```
funMethod("Hello");
```

This is a very common error to see in Java.

- You can turn the method static by adding the static keyword in the method definition *(Easy and quick fix)*
- Or you use OOP and call the method on an instance of the class

```
AnotherClass obj = new AnotherClass()
obj.funMethod("Hello")
```

*(Usually this is the better solution 80% of the time)*

**Abstract Classes** are restricted classes that cannot be used to create objects. To access it, it must be inherited from another class.

```
public abstract class Employee {
    ...
}
```

```
Employee e = new Employee("Sally", 4444, 123456);
```

You **cannot** create instances of an abstract class.

```
Accountant kevin = new Accountant("Kevin Malone", 4444, 42000, 'C');
```

Instead, we use objects from another class *that inherits from the abstract class*

**Polymorphism** is the ability of a class to provide different implementations of a method, depending on the *type of object* that is passed to the method.

```
Bird a2 = new Bird("Puffin",27.0, "North America",7400000,21.5);
Wolf b2 = new Wolf("Arctic Wolf",120.0, "North America",200000, 16);

a2.makeSound();
b2.makeSound();
```

The `makeSound()` method does something different for each object

**Polymorphism** is the ability of a class to provide different implementations of a method, depending on the *type of object* that is passed to the method.

*Polymorphism also refers to the ability for an object to take many forms*

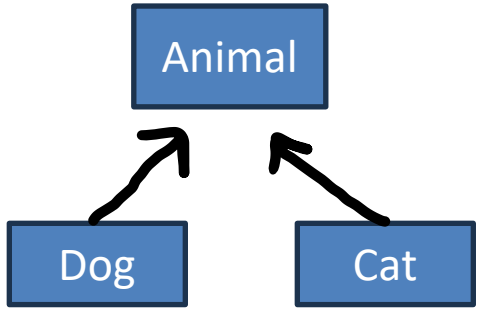
```
Programmer me = new Programmer(“Reese", 12345, 80000, “Python”);

Employee me = new Programmer(“Reese", 12345, 80000, “Python”);
```

We can also treat the `me` reference variable as an `Employee`, since `Programmer` inherits from `Employee`

```
Dog tater = new Dog();
Cat meatball = new Cat();

Animal[] myAnimalArray = {tater, meatball};
```



**`try/catch`** and **`exceptions`** are a way to run a piece of code (“try”), and then deal (“catch”) with errors

It will execute the body of **`try`**, and if a certain error/exceptions arises, then it will run the body of the **`catch`** statement

You can catch any error, or a specific error (FileNotFound, ArrayIndexOutOfBounds, NullPointerException)

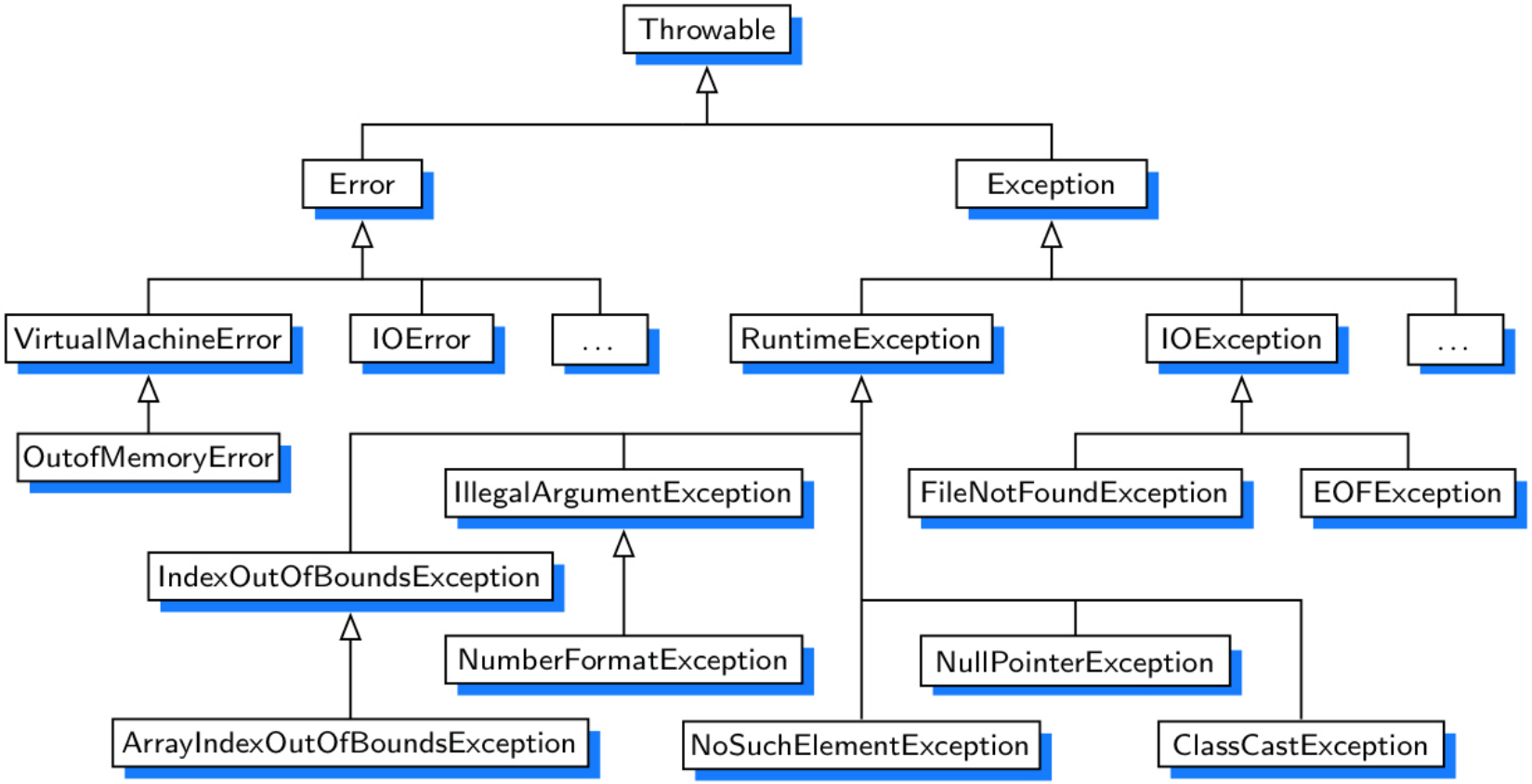



**Figure 2.7:** A small portion of Java's hierarchy of Throwable types.

```
while(userChoice != 3) {
        try {
                Scanner scanner = new Scanner(System.in);
                userChoice = scanner.nextInt();
                if(userChoice == 1) {
                        System.out.println("Hello");
                }
                else if(userChoice == 2) {
                        System.out.println("Hey");
                }
                else if(userChoice == 3) {
                        System.out.println("Goodbye");
                }
                else {
                        System.out.println("Enter a valid integer");
                }
        }
        catch(Exception e) {
                System.out.println(e);
                System.out.println("Invalid input detected. Please try again");
        }
        printOptions();
```

Java will **try** to execute this block of code

```
while(userChoice != 3) {
        try {
                Scanner scanner = new Scanner(System.in);
                userChoice = scanner.nextInt();
                if(userChoice == 1) {
                        System.out.println("Hello");
                }
                else if(userChoice == 2) {
                        System.out.println("Hey");
                }
                else if(userChoice == 3) {
                        System.out.println("Goodbye");
                }
                else {
                        System.out.println("Enter a valid integer");
                }
        }
        catch(Exception e) {
                System.out.println(e);
                System.out.println("Invalid input detected. Please try again");
        }
        printOptions();
```

Java will **try** to execute this block of code

If any error happens, instead of crashing, it will execute the body of the **catch**

# Debugging Code

Our IDE has a super slick debugger built in to it. I highly recommend learning how to use the debugger tool (see lecture)

## Rubber Duck Debugging

Many programmers have had the experience of explaining a problem to someone else, possibly even to someone who knows nothing about programming, and then hitting upon the solution in the process of explaining the problem. In describing what the code is supposed to do and observing what it actually does, any incongruity between these two becomes apparent.[2] More generally, teaching a subject forces its evaluation from different perspectives and can provide a deeper understanding.[3] By using an inanimate object, the programmer can try to accomplish this without having to interrupt anyone else, and with better results than have been observed from merely thinking aloud without an audience.

*(From Wikipedia)*