Polymorphism in Java – Contd. Abstraction & Interfaces!

CS 171: Intro to Computer Science II

Me: *explains polymorphism*

Friend: So the subclass the same thing as the superclass?

Me:

Reminder: A1 due this Wed 2/15

- Due Wed 2/15 at 11:59PM
  !! Remember !! if you submit even a second after 11:59pm it will be considered a late submission; you will use your late-token!

- Submit early to Gradescope to make sure your code compiles and passes test-cases

- Review Submission Checklist at the end of A1 handout!

Polymorphism: Review

Given that class `SalariedEmployee` extends class `Employee`, which of the following are allowed in Java?

A. `Employee emp = new Employee("Kiko");`
   `SalariedEmployee sEmp = (SalariedEmployee) emp;`
B. `SalariedEmployee sEmp = new SalariedEmployee("Ron", 100.0);`
   `Employee emp = sEmp;`
C. `SalariedEmployee sEmp = new SalariedEmployee("Ron", 100.0);`
   `Employee emp = (Employee) sEmp;`
D. A and B
E. B and C
F. A and C
G. A, B, and C

pollev.com/elsayedcs171

Polymorphism: Review

Given that class `SalariedEmployee` extends class `Employee`, which of the following are allowed in Java?

A.
```
Employee emp = new Employee("Kiko");
SalariedEmployee sEmp = (SalariedEmployee) emp; // narrowing
                                                   conversion
```

B.
```
SalariedEmployee sEmp = new SalariedEmployee("Ron", 100.0);
Employee emp = sEmp;  // widening conversion
```

C.
```
SalariedEmployee sEmp = new SalariedEmployee("Ron", 100.0);
Employee emp = (Employee) sEmp; // widening conversion
```

D. A and B
E. B and C ✓
F. A and C
G. A, B, and C

Dynamic Dispatch

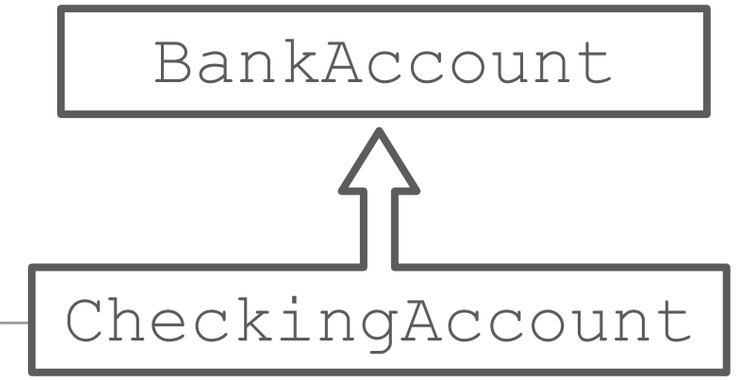
- What about methods of the `CheckingAccount` class that override methods of the `BankAccount` class? Which version is used?

- *Dynamic dispatch*: Java makes a runtime decision to call the method version associated with the actual type of the referenced object (not the declared type)

```
BankAccount x = new CheckingAccount();
```

Q: which version of method `deposit` will `x.deposit(100)` invoke?
Answer: The `CheckingAccount` version!

instanceof



- Operator that tests at runtime if an instance satisfies a particular type

```
BankAccount x = new BankAccount();
boolean b = x instanceof BankAccount; //true
b = x instanceof CheckingAccount; //false

x = new CheckingAccount();
b = x instanceof CheckingAccount; //true
b = x instanceof BankAccount; //true!
```

Rule: `x instanceof ClassName` evaluates to `true` if `x` references an object belonging to the `ClassName` class or any further subclass of `ClassName`

Q from last week: Why does this narrowing conversion generate an exception?

```
Employee emp = new Employee("Kiko");
SalariedEmployee sEmp = (SalariedEmployee) emp;
// narrowing conversion (not allowed)
```

Recall that a SalariedEmployee is instantiated with a specific name and a salary. What's the salary for "emp" in this case?

See Code Example (all scenarios included): PolymorphicEmployee.java

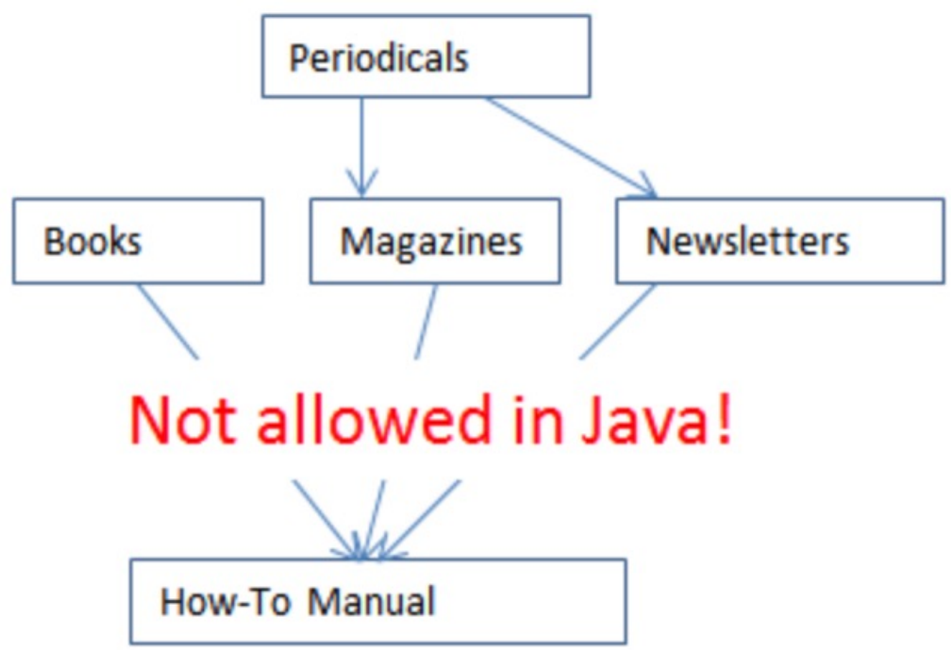
Me: *explains polymorphism*

Friend: So the subclass the same thing as the superclass?

Me:

Limitations of Inheritance

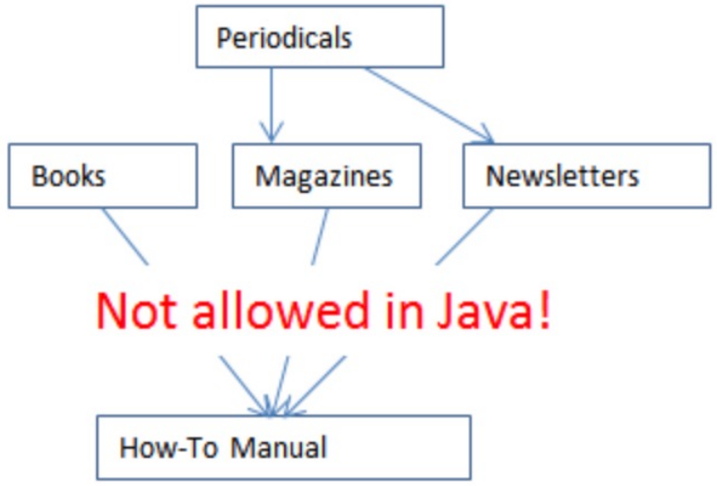
- Can I inherit from more than one parent class?



Inheritance: Limitations [1/2]

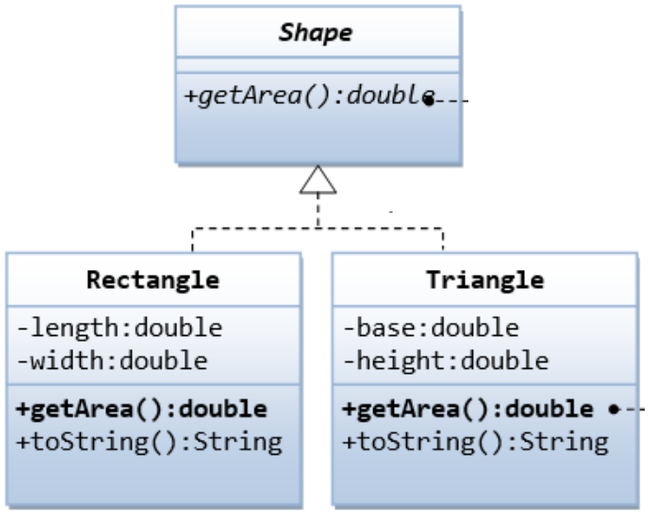
- What if I want to inherit from more than one parent class?
  - In biology: possible

  - In Java: prohibited
  - You can’t inherit from multiple classes



Inheritance: Limitations [2/2]

- What if we don't want to allow creation of superclass objects?
- E.g.: Bank doesn't want customers to open a general BankAccount!
- Or, only *particular* shapes makes sense, not some *generic* shape



Shape sh = new Shape();

Rectangle re = new Rectangle();

Triangle tr = new Triangle();

Solution: "Abstract" Classes!

Review: Object-Oriented Programming

First: Abstract methods

- In Java: You can declare a method *without* defining it

```
public abstract double getArea( );
```

➔ Notice the body of the method { … } is missing!

- A method that has been declared but not *defined* is an abstract method

Abstract class

- Any class containing *one or more* abstract methods is an abstract class

- You must declare the class with the keyword **abstract**:

```
abstract class MyClass {...}
```

- An abstract class is incomplete!

  ➔ Has “missing” method bodies

  ➔ You cannot instantiate (create a new instance of) an abstract class

Why have abstract classes?

- You can **extend** (subclass) an abstract class
  - If subclass defines all inherited abstract methods, it is "complete" and can be instantiated

```
CompleteSubClass sc = new CompleteSubClass ();
```

  - If subclass does not define all inherited abstract methods, it too must be abstract (& can't be instantiated!)

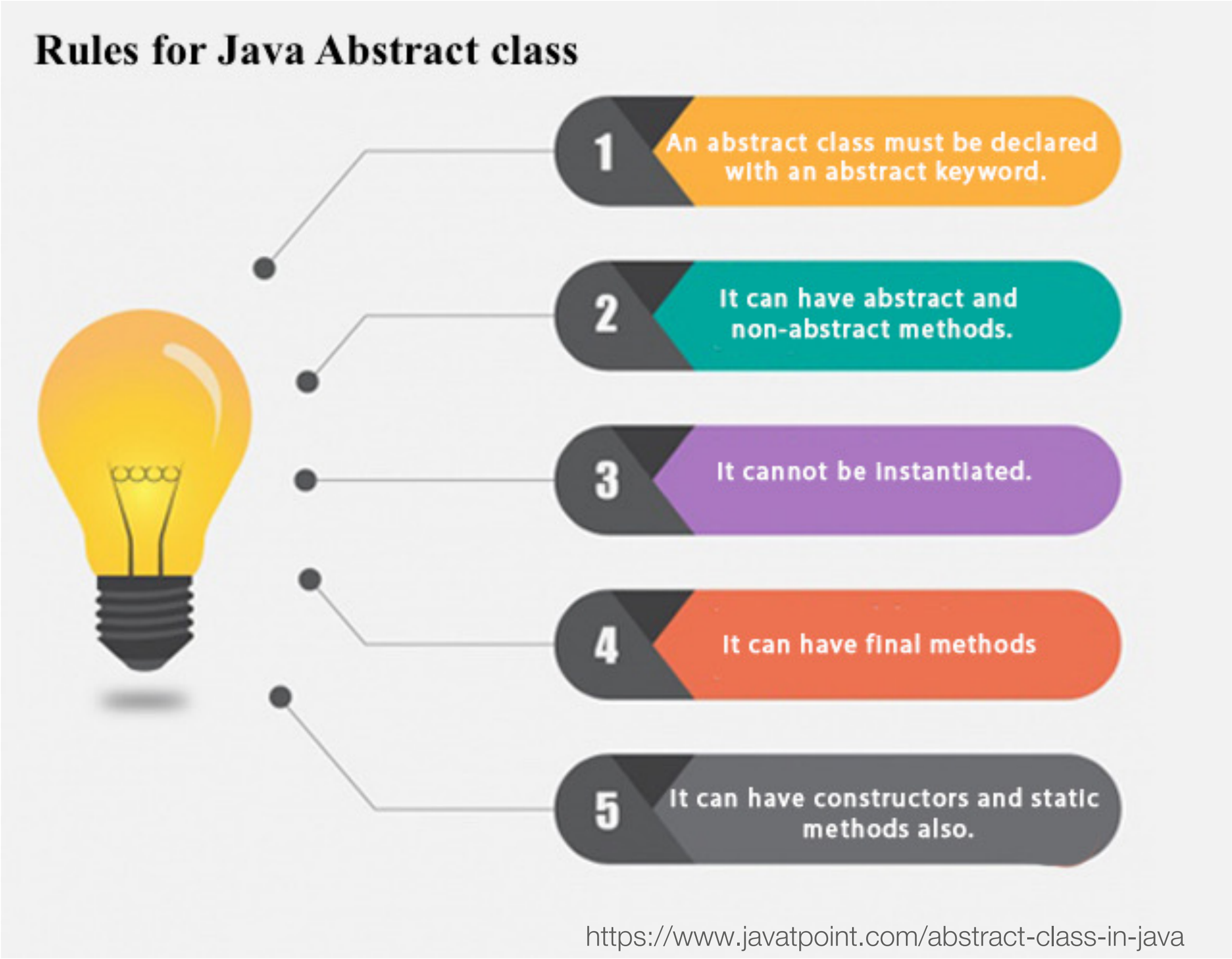
It gets more interesting…

- You can declare a class to be **abstract** even if it doesn't contain any abstract methods!

```
SomeAbstractClass sc = new SomeAbstractClass ();
```

  ➔ Prevents the class from being instantiated!

Think about using abstract classes when something needs to be there but not exactly sure how objects should look


Rules for Java Abstract class
1
An abstract class must be declared with an abstract keyword.
2
It can have abstract and non-abstract methods.
3
It cannot be Instantiated.
4
It can have final methods
5
It can have constructors and static methods also.
https://www.javatpoint.com/abstract-class-in-java

Example: An Abstract Car

Cannot instantiate an AbstractCar object!

```java
public abstract class AbstractCar{
    private String brand;
    private String model;
    private String color;
    private double mileage;

    public AbstractCar(String b, String m, String c){
        brand = b;
        model = m;
        color = c;
        mileage = 0;
    }

    public String toString(){
        return brand + " model " + model + " in " + color;
    }

    public void run(double miles){
        mileage += miles;
    }

    public double getMileage(){
        return mileage;
    }

    public abstract void driveSelf(double miles);
}
```

Honda: Extending AbstractCar

```java
public class Honda extends AbstractCar {
    public Honda(String model, String color) {
        super("Honda", model, color);
    }

    public void driveSelf(double miles) {
        System.out.println("Driving an awesome Honda");
        run(miles);
        System.out.println("Number of miles on the car:" + getMileage());
    }

    public static void main(String[] args) {
        AbstractCar accord = new Honda("Accord", "Green");
        AbstractCar crv = new Honda("CRV", "Blue");

        accord.driveSelf(100);
        crv.driveSelf(20);
    }
}
```

```java
public abstract class AbstractCar {
    private String brand;
    private String model;
    private String color;
    private double mileage;

    public AbstractCar(String b, String m, String c) {
        brand = b;
        model = m;
        color = c;
        mileage = 0;
    }

    public String toString() {
        return brand + " model " + model + " in " + color;
    }

    public void run(double miles) {
        mileage += miles;
    }

    public double getMileage() {
        return mileage;
    }

    public abstract void driveSelf(double miles);
}
```

```java
public class Honda extends AbstractCar {
    public Honda(String model, String color) {
        super("Honda", model, color);
    }

    public void driveSelf(double miles) {
        System.out.println("Driving an awesome Honda");
        run(miles);
        System.out.println("Number of miles on the car:" + getMileage());
    }

    public static void main(String[] args) {
        AbstractCar accord = new Honda("Accord", "Green");
        AbstractCar crv = new Honda("CRV", "Blue");

        accord.driveSelf(100);
        crv.driveSelf(20);
    }
}
```

- Q: Can `AbstractCar` be instantiated?
  - No (it's an abstract class)
- Can `Honda` be instantiated?
  - Yes

Practical Usages of Abstract Classes

1. You are unsure of how a method should be defined/ implemented for that class

2. You don’t want the possibility of objects of that type being created (e.g. bank doesn’t want customers creating a general bank account)

pollev.com/elsayedcs171

Question

Can a Java class be abstract and final?

A. Yes

✓ B. No

pollev.com/elsayedcs171

Question

Can a Java method be abstract and final?

A. Yes

✓ B. No

What if <u>all</u> methods in a class are abstract?

Declare it as <u>**interface**</u>

Interface

- Implies 100% abstraction (no implemented methods)
- Reference type similar to class (but not a class!)
- Collection of abstract methods or group of related methods with empty bodies
- Can contain variables but must be static and final (can't be changed)
- Cannot be instantiated or contain any constructors

Example: Shape Interface

```
interface Shape{
    public double getArea();

    public int getNumSides();
}
```

Interfaces tell the world: what is the functionality (behavior) that should be offered -- irrespective of underlying implementation!

Square implements Shape

You **extend** a *class*, but you **implement** an *interface*

```java
public class Square implements Shape{
    private static final int numSides = 4;
    private double sideLength;

    public Square(){
        sideLength = 1.0;
    }

    public Square(double sideLength){
        this.sideLength = sideLength;
    }

    public double getNumSides(){
        return numSides;
    }

    public double getArea(){
        return sideLength * sideLength;
    }

    public double getPerimeter(){
        return 4*sideLength;
    }
}
```

implements = signing a binding contract!

- When you say a class **implements** an interface, you are promising to **define** all the methods that were **declared** in the interface!

Why interfaces?

- A class can only **extend** one other class, but it can **implement** multiple interfaces
  - This lets the class fill multiple “roles”
  - Example:

```
class MyApp extends App
        implements ActionListener, KeyListener {
  ...
} // Now you must implement ALL methods in both ActionListener and KeyListener!
```

instanceof

- instanceof is a keyword that tells you whether a variable "is a" member of a class or interface
- For example, if

```
class Dog extends Animal implements Pet {...}

Dog fido = new Dog();
```

which of the following is true?

```
fido instanceof Dog
fido instanceof Animal      // superclass!
fido instanceof Pet         // interface!
```

(Answer: ALL of them are true!)

Abstract Class vs. Interface

- Abstract Classes can have constructors, instance variables, & concrete methods (i.e. non-abstract); *interfaces can't!*

- Abstract Classes are *extended* by a subclass which **may** implement the abstract methods, but doesn't have to.

- Interfaces are *implemented* by another class which **must** implement its abstract methods.

pollev.com/elsayedcs171

Question

A class can implement more than one interface.

✓ A. True

B. False

pollev.com/elsayedcs171

Question

A class **ClassA** can extend an interface **ClassB** by declaring that, and by implementing any abstract methods in **ClassB**.

A. True

✓ B. False

Review of some OOP concepts

- A class is like a definition of a type in Java.
- An instance of an object of a given class can be created using a constructor and the `new` operator.
- Instances are accessed through reference variables.
- A subclass can extend a superclass and use its methods and instance variables through inheritance and polymorphism.
- Abstract classes contain at least one abstract method and must be extended by a subclass that defines the abstract methods.
- Interfaces are 100% abstraction and provide a template for another class to implement all of the methods defined in the interface.

Let’s talk about Exceptions

- Unhandled exceptions cause the code to terminate
- Reminder: how to handle Exceptions

```
try{
    // block of code that may throw exception
}
catch(Exception e){
   // deal with exception here
}
```

Exceptions Hierarchy

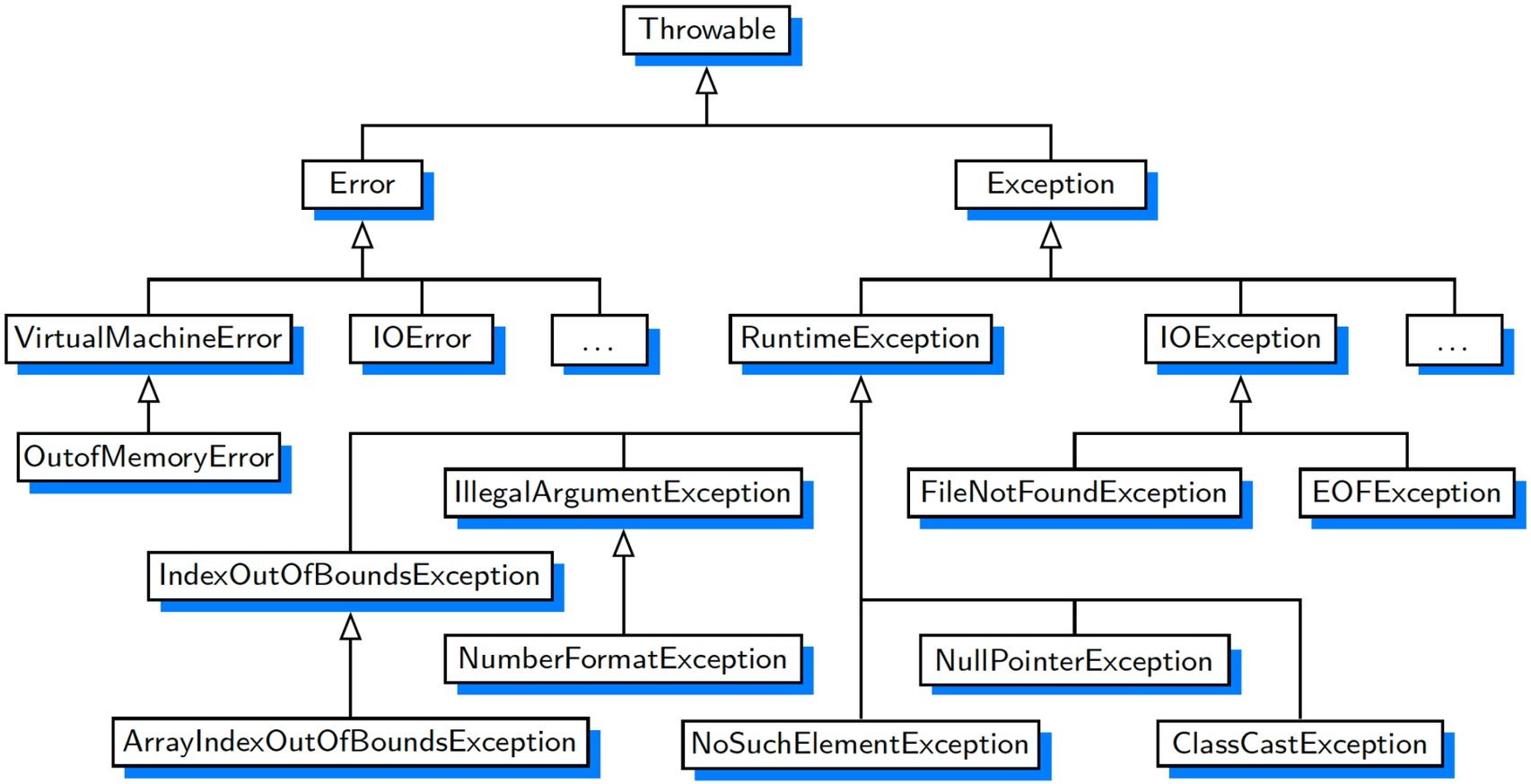


Figure 2.7: A small portion of Java's hierarchy of Throwable types.

Example: IndexOutOfBoundsException

java.lang

Class IndexOutOfBoundsException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IndexOutOfBoundsException
```

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException

---

Exceptions & Polymorphism

- Can catch a more general exception
- “Catch-all:” use Exception itself

```
try {
    System.out.println(myNumbers[10]); // throws ArrayIndexOutOfBoundsException!
}
catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Error! Bad index: " + e.getMessage());
}
catch(Exception e){
    System.out.println("Some other error occurred: " + e.getMessage());
}
```