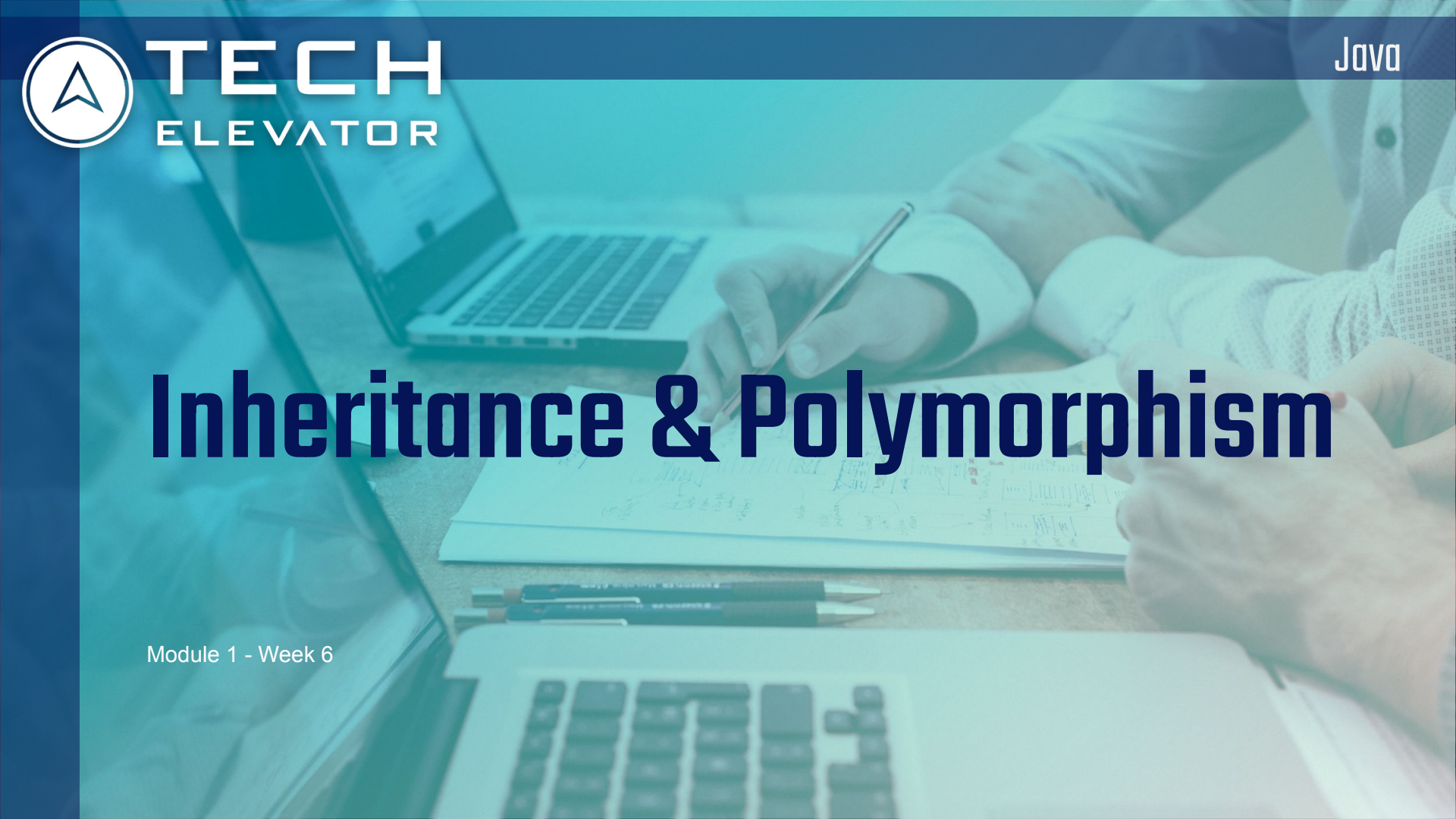


TECH ELEVATOR

Java

# Inheritance & Polymorphism

Module 1 - Week 6

- What are the 3 (or 4?) pillars of Object Oriented Programming.
- What are some advantages of using Object Oriented Programming?
- What is Encapsulation?
- What is Polymorphism?

# Housekeeping

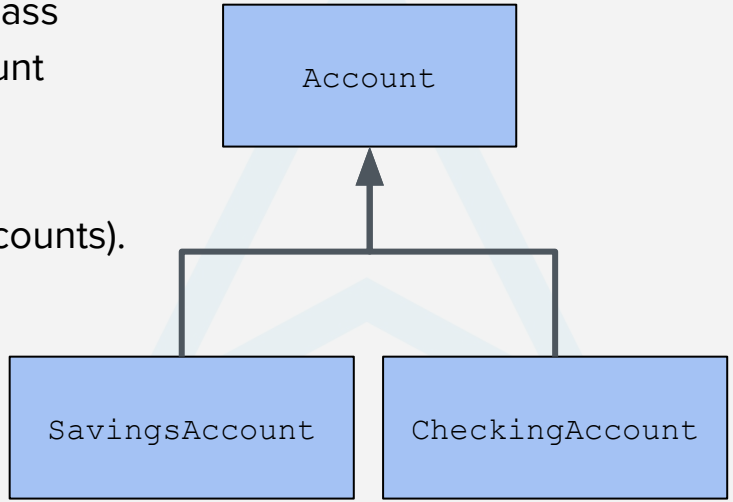
- **Extra practice:** I added coding katas to your student code repos.
- **Week 7:** there are going  to be two units in the LMS next week, but only one (Unit Testing) has exercises. The unit on Exception Handling only has a tutorial, but no exercises.

# Schedule


| Week | Topics |
|---|---|
| Week 0 | Welcome, Intro to Tools |
| Week 1 | Variables and Data Types, Logical Branching |
| Week 2 | Loops and Arrays, Command-Line Programs, Intro to Objects |
| Week 3 | Collections |
| Week 4 | **Mid-Module Project** |
| Week 5 | Classes and Encapsulation |
| Week 6 | Inheritance, Polymorphism |
| Week 7 | Unit Testing, Exceptions and Error Handling |
| Week 8 | File I/O Reading and Writing |
| Week 9 | **End-of-module project** |
| Week 10 | **Assessment** |

# Inheritance

- **Inheritance**: the practice of creating a hierarchy for classes in which descendants obtain the attributes and behaviors from other classes
- Can be described as:
  - Parent / Child relationship
  - Superclass / Subclass relationship
  - Base / Derived relationship
- Derived classes are specializations of a base class
  - A savings account **is a** type of bank account
  - A reserve auction **is a** type of auction
- Referred to as an **“is-a” relationship**
- Goes **one way** (not all accounts are savings accounts).

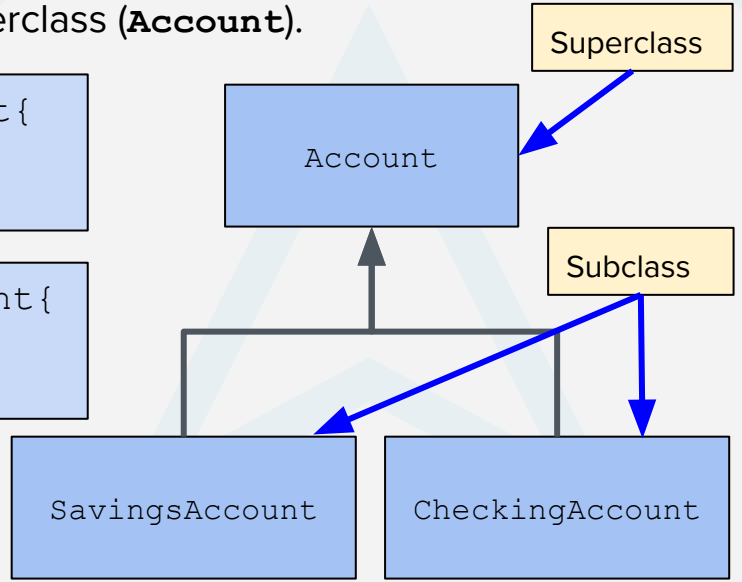


- Java provides powerful tools that enable developers to model these parent-child relationships.
- A class can be designated as a child of a parent class and inherit from that class using the **`extends`** keyword.
- Once extended, the subclasses (**`SavingsAccount`** & **`CheckingAccount`**) will inherit all non-private properties and methods from the superclass (**`Account`**).

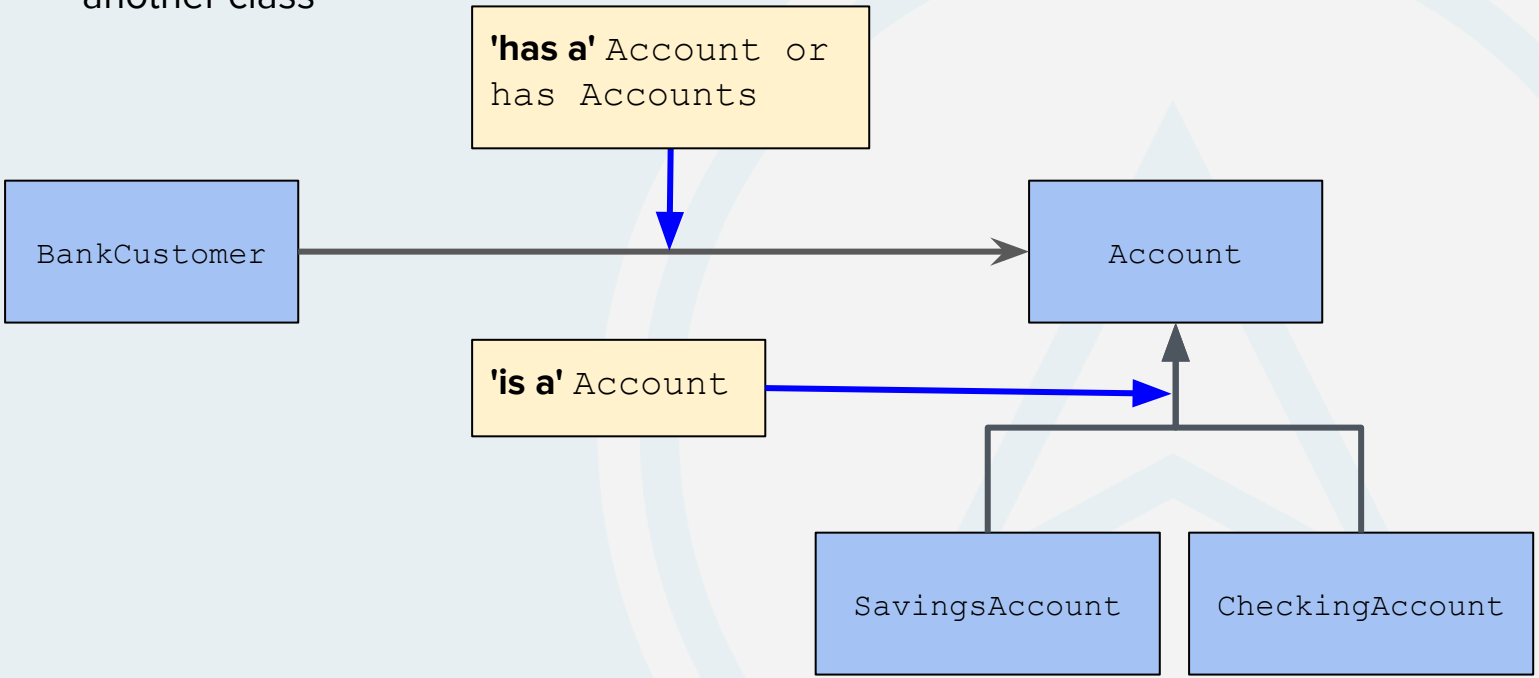
```
public class SavingsAccount  extends Account{
    // code here…
}
```

```
public class CheckingAccount  extends Account{
    // code here…
}
```

**'is a'** Account

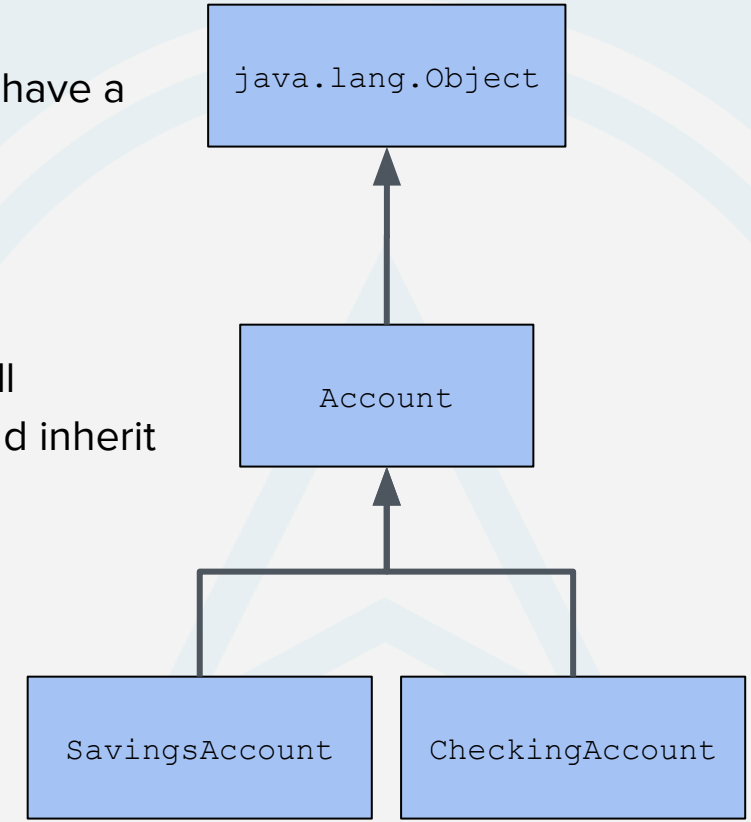


- **Is-a** relationship depends on inheritance.
- **Has-a** relationship is also known as composition.
  - an instance of one class has a reference to an instance of (or collection of) another class

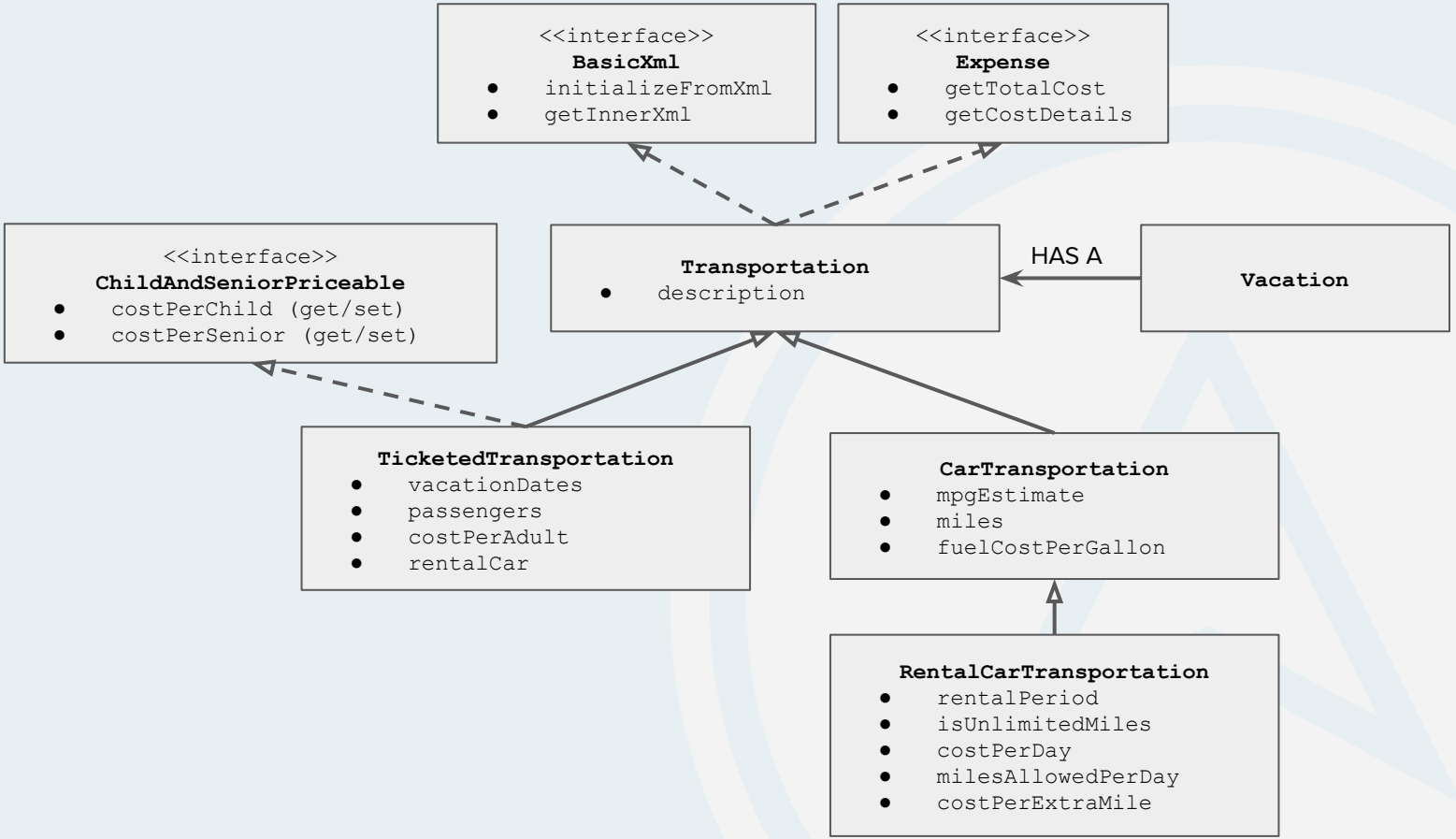


# java.lang.Object

- In Java, all objects (reference types) are subclasses of the class **`java.lang.Object`**.
- **`Object`** is the only class in Java that does not have a superclass.
- The only things in the language that are not descendents of **`java.lang.Object`** are the primitives: long, int, double, boolean, etc.
- Even if no superclass is specified, all classes still implicitly extend from **`java.lang.Object`**, and inherit a set of common methods, such as:
  - `toString()`
  - `equals()`
  - `hashCode()`



# Transportation Hierarchy



# Overriding

- A subclass can override a method from the superclass by redefining the method.
- When a method is called on an instance of a subclass,  the subclass version of the method will be called if defined, otherwise the superclass method will be.
- The method signature must match the signature being overridden exactly.
- Java provides the **`@Override`** annotation to note that a method overrides a super method.
- If you use the **`@Override`** annotation on a method you intend to override, you will get a compiler error if your signature does not match the signature of any signatures in the superclass. This is very useful to ensure your method WILL actually override as intended.
- If a subclass overrides a superclass method, that class can always call the superclass method by using the **`super`** prefix to access the super version of the method.

```
public class Account{
  public int withdraw(int amount){
    // code here…
  }
}
```

```
public class SavingsAccount extends Account{
  @Override
  public int withdraw(int amount){
    // code here…
  }
}
```

# Overriding toString

We can override the methods that we inherit from `java.lang.Object`

```
CheckingAccount account = new CheckingAccount("Tom", "123456", 1000000);
System.out.println(account);
```

```
com.techelevator.CheckingAccount@57829d67
```

```
public class CheckingAccount extends Account{
  @Override
  public String toString() {
    return "BankAccount{accountHolderName='" + getAccountHolderName() +
           '\'' + ", accountNumber='" + getAccountNumber() + '\'' +
           ", balance=" + getBalance() +'}';
  }
  …
}
```

```
BankAccount{accountHolderName='Tom', accountNumber='123456',
balance=1000000}
```

# Super Constructors

- A subclass constructor must call a superclass constructor if the superclass does not expose the default no-arg constructor.
- A subclass constructor calls its superclass constructor using the **`super`** keyword..
- Constructors are not inherited and must always be invoked using **`super`**.

```
public class Vehicle{
    private int wheels;
    public Vehicle(int wheels) {
        this.wheels = wheels;
    }
}
```

```
public class Car extends Vehicle{
    private String color;
    public Car(int wheels, String color) {
        super(wheels);
        this.color = color;
    }
}
```

**The `Car` constructor must call a `Vehicle` constructor.**

# Overloading Constructors

- Classes can contain more than one constructor, each taking a different number or data types of arguments.
- We are overloading the constructor (just like we can overload any other method)
- Classes can chain constructors by using **`this`** to call another overloaded constructor:

```
public class Car extends Vehicle{
    private String color;
    public Car(String color) {
        this(4, color);
    }

    public Car(int wheels, String color) {
        super(wheels);
        this.color = color;
    }
}
```

**One Car constructor can chain to another `Car` constructor.**

# Define what overriding means in the context of inheritance - Objective 2

**Some common difficulties with overriding in the context of inheritance...**

- not understanding the value of the `@Override` annotation
- confusing **overriding** and **overloading**
- using `super` to call the method you're overriding

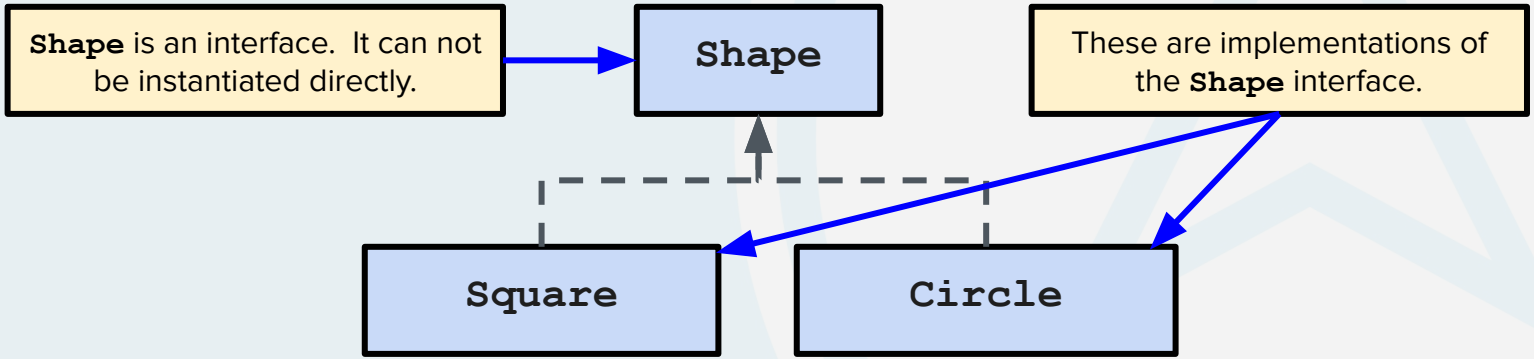
- **Polymorphism** is the ability of an object to take on many forms. The most common use of polymorphism is when a parent class reference is used to refer to a child class object.
- **Polymorphism using inheritance** is the concept that any object which is a subclass can be treated as the superclass type.

```
List<Account> accounts= new ArrayList<>();

accounts.add(new SavingsAccount(1000));
accounts.add(new CheckingAccount(2000));
…
for ( Account account: accounts) {
    BigDecimal balance = account. getBalance();
    …
    account.withdraw(new BigDecimal(20.00));
}
```

# Interfaces

- An **interface** specifies behavior (methods) without implementation (body).
- Creates a data type that can be used to declare variables but can't be instantiated directly.
- An interface is a **contract** that defines which methods a user of the interface can expect.
- Keyword: **`implements`**
- A class may implement more than one interface.
- An interface can be implemented by multiple concrete classes. These concrete classes provide different implementations of the abstract methods defined by the interface.
- If class `Square implements` the interface `Shape`, then `Square` "**is-a**" `Shape`.



```
public interface Shape{
    void draw();
    double getArea();
}
```

No access modifier specified (all interface methods are **public**).

Ends with semicolon because there is no code. Interface is a contract, not implementation.

The class implements the interface.

The class provides implementations for the interface methods.

```
public class Square implements Shape {

    private final double length;

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

    @Override
    public void draw() {
        // draw code here...
    }

    @Override
    public double getArea() {
        return length * length;
    }
}
```

# Polymorphism with Interfaces

- Polymorphism can also be implemented using **interfaces**.
- An interface creates a data type to which the object can be cast. This allows objects with the same interfaces to be grouped generically, while still providing their specific response when a method is invoked.

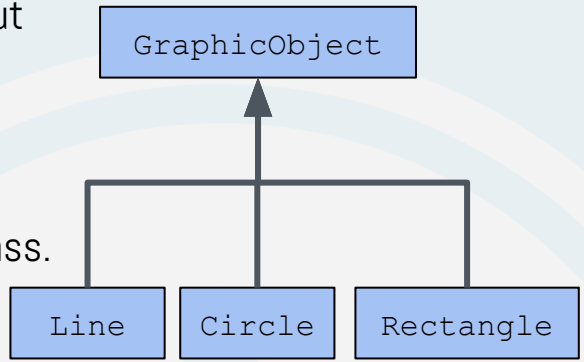
```
List<Shape> shapes = new ArrayList<>();

shapes.add( new Square(9.2) );
shapes.add( new Rectangle(10.0,8.1) );
shapes.add( new Circle(7.6) );

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

# Abstract

- Abstract classes can not have objects created from them, but they can provide logic and structure to their subclasses.
- Abstract methods are methods with no logic that must be implemented by concrete subclasses.
- If a class has an abstract method, it must be an abstract class.
- If a class does not implement an abstract method from its parent, it must also be an abstract class




```
abstract class GraphicObject {
    private int x, y;
    ...
    void moveTo(int newX, int newY)
{
        x = newX; y = newY;
    }
    abstract void draw();
    abstract void resize();
}
```

```
class Circle extends GraphicObject {
    void draw() {
        // draw code here...
    }

    void resize() {
        // resize code here...
    }
}
```

# Interface vs Abstract Class

Interface:

- An interface is a reference type in Java that defines a contract for its implementing classes.
- It contains abstract method signatures, which means the methods declared in the interface have no implementation and are meant to be implemented by the classes that implement the interface.
- Interfaces support multiple inheritance, as a class can implement multiple interfaces.
- Interfaces are used to achieve abstraction, decoupling, and provide a way to enforce a specific set of behaviors on implementing classes.

Abstract Class:

- An abstract class is a class that cannot be instantiated, meaning you cannot create objects directly from an abstract class.
- It can have both abstract and non-abstract methods.
- Abstract methods in an abstract class also have no implementation and must be implemented by its concrete subclasses.
- Abstract classes support single inheritance, meaning a class can extend only one abstract class.
- Abstract classes are used to provide a base or common functionality for multiple related classes, and they may also include some default implementations for methods.

# Interface vs Abstract Class

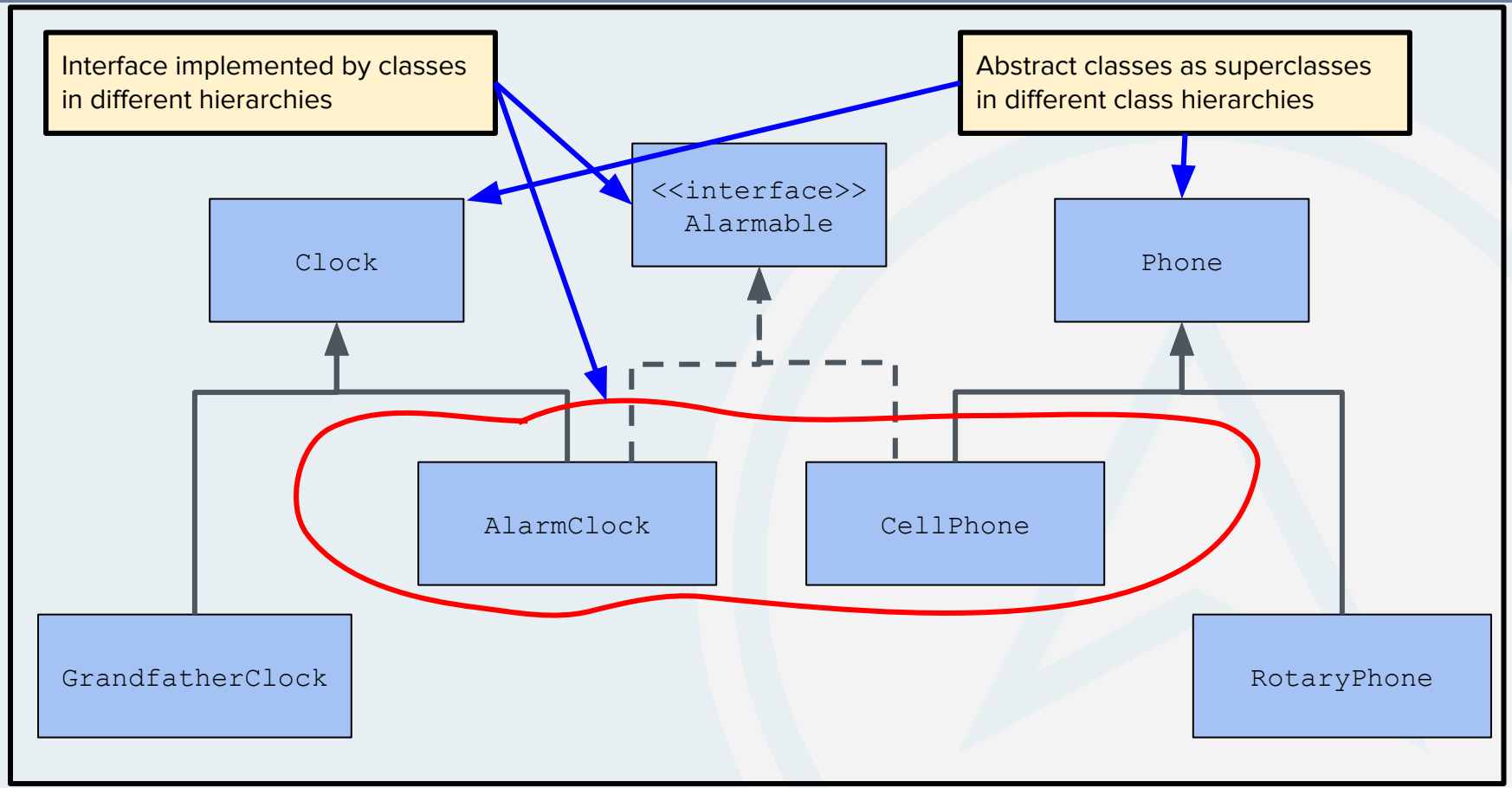
When to use an Interface:

- When you want to define a contract that enforces specific behaviors on implementing classes. For example, defining a Runnable interface to indicate that classes can be executed in a separate thread.
- When you need to support multiple inheritance, as a class can implement multiple interfaces, allowing for more flexibility in class design.
- When you want to create a loosely coupled design, enabling different implementations for the same behavior, such as using different implementations of a DataSource interface for various database types.

When to use an Abstract Class:

- When you want to provide a common base with some default functionality that multiple related classes can inherit and build upon.
- When you need to define fields or non-static methods that can be shared among the subclasses.
- When you want to have the flexibility of adding abstract and non-abstract methods in the same class.

# Interface vs Abstract Class



# Use polymorphism through interfaces using IS-A relationships - Objective 3

**Some common difficulties with polymorphism and interfaces...**

- confusion about when to use an **interface vs. inheritance**
- mistakenly putting code in interfaces