Brighter Blue

Advanced OOP with Examples

Core Java Day 2

IBM

Agenda

Brighter Blue

Day 02

01 Abstraction and Encapsulation

03 Polymorphism

02 Inheritance

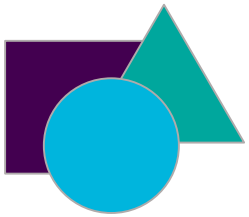
Brighter Blue

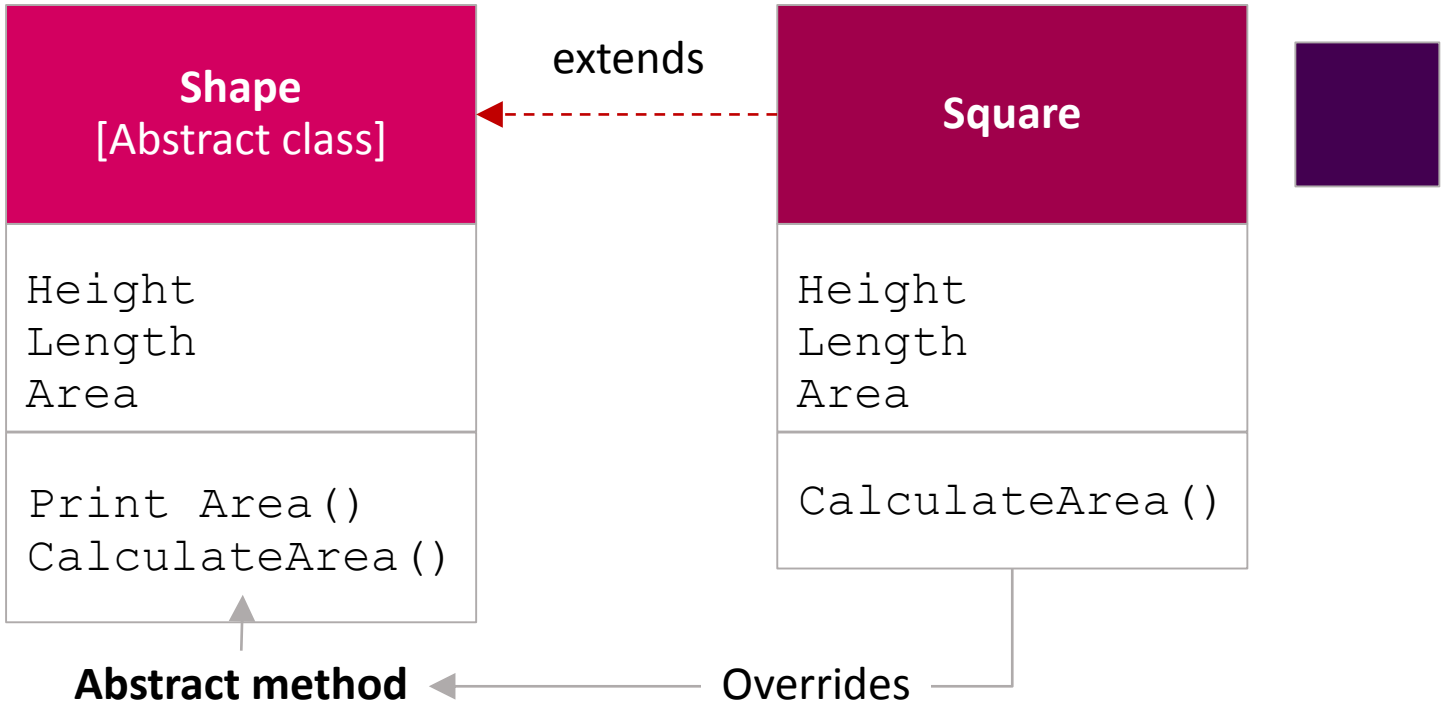
01 Abstraction and Encapsulation

IBM

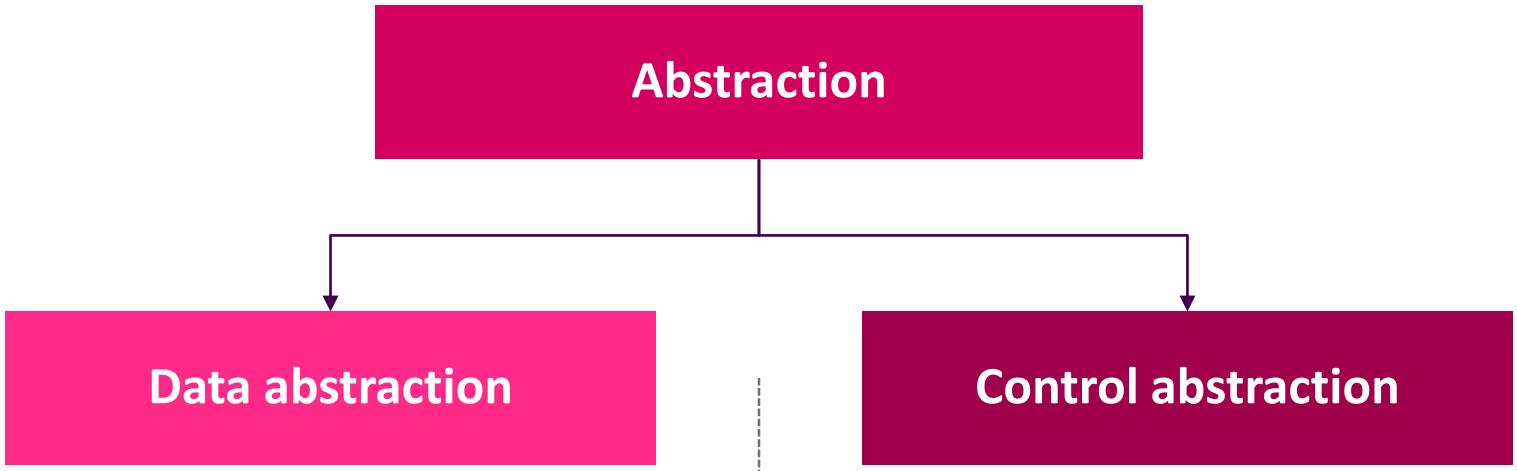
What is **abstraction**?

Abstraction involves the facility to define objects that represent abstract “actors” that can perform work, report on and change their state, and “communicate” with other objects in the system.





Types of **abstraction**



- Used to create complex data types
- Exposes only meaningful operations to interact with the data
- Hides all implementation details from outside works.

- Defines new control constructs
- Specifies a statement ordering separately from an implementation of that ordering
- Plays a central role in parallel programming.

Activity

Brighter Blue

Demonstration: **Abstraction**

```
public abstract class Shape {
    public abstract double area();
    public abstract double perimeter();
}
public class Rectangle extends Shape {
    private final double width, length; //sides
    public Rectangle() {
        this(1,1);
    }
    public Rectangle(double width, double length) {
        this.width = width;
        this.length = length;
    }
    public double area() {
        // A = w * l
        return width * length;
    }
    public double perimeter() {
        // P = 2(w + l)
        return 2 * (width + length);
    }
}
public class Main {
  public static void main(String args[]) {
    Rectangle r = new Rectangle(10, 20);
System.out.println("Area :" +r.area());
System.out.println("Perimeter :" +r.perimeter());
}
```


IBM

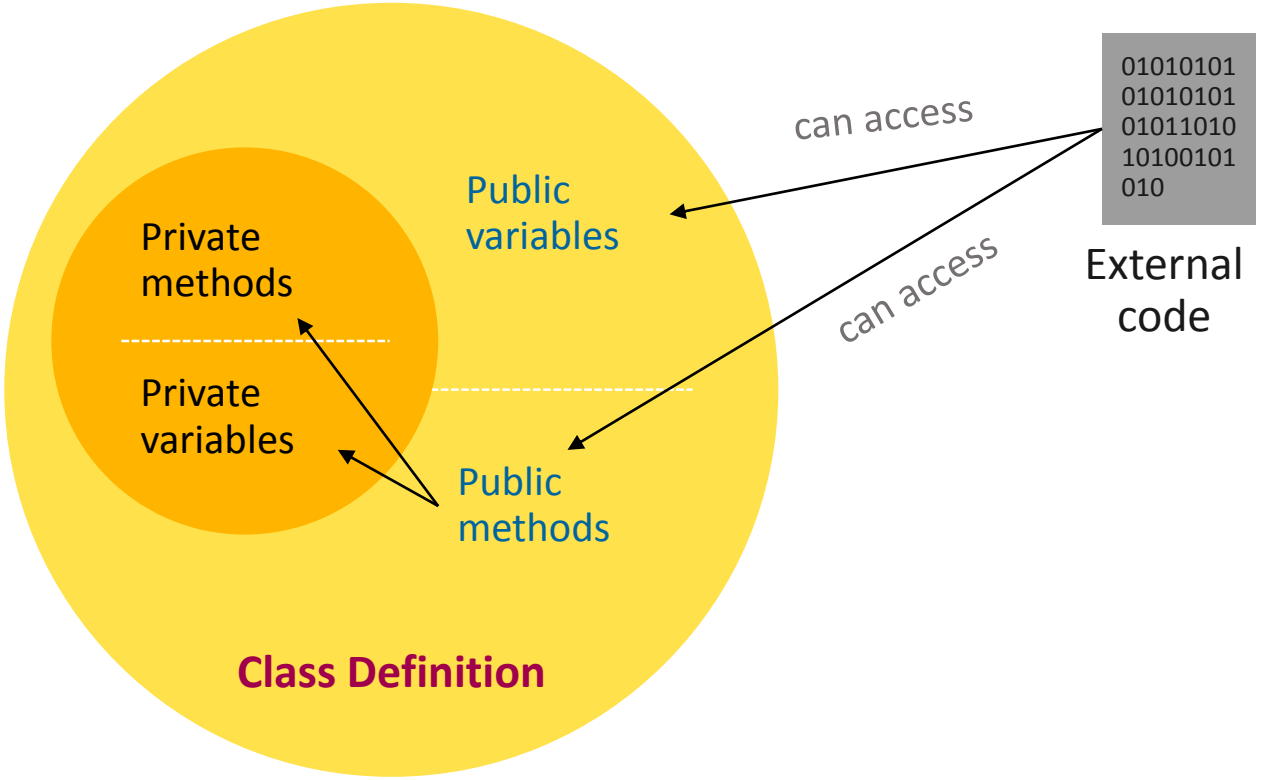
Abstraction: example

```
abstract class Car{

abstract double getOnRoadPrice();
}

class Honda extends Car{
double getOnRoadPrice(){return 850000;}
}
class Toyota extends Car{
double getOnRoadPrice(){return 950000;}
}

class CarMain{
public static void main(String args[]){
    Car h=new Honda ();
    double onRoadPrice = h.getOnRoadPrice();
    System.out.println("On Road Price of Honda: "+onRoadPrice);
    }
}
```

What is **encapsulation**?

Brighter Blue

Encapsulation, is the idea that the **data** associated with an object should (mostly) only be available via functions, and (possibly) that much of the data associated with an object will never be **visible** to the outside world.



IBM

Activity

Brighter Blue

Demonstration: **Encapsulation**

Private variables

```
private String userName;
private String password;
private String FirstName;
private String LastName;
private String email;
```

Public methods

```
public int getUserName(){
      return userName;
   }
   public String getPassword(){
      return password;
   }
   public String getFirstName(){
      return Firstname;
   }
   public String getLastName(){
      return LastName;
   }
   public String getEmail(){
      return email;
   }
   public void setUserName( String uName){
      userName= uname;
   }
   public void setPassord(String uPass){
      password= uPass;
   }
   public void setFirstName(String UFirst){
      FirstName= uFirst;
   }
   public void setLastName(String ULast){
      LastName= uLast;
   }
   public void setEmail( String emailId){
      email= emailId;
   }
}
```

Class **User**

Encapsulation: example

Private data members can be accessed using public methods.

```
public class UserEncapImpl{

   public static void main (String args[]){
      User user= new User ();
      user.setUserName ("IBM");
      user.setPassword ("*****");
      user.setEmail ("ibm@us.ibm.com");

      System.out.println ("Name: " + user.getUserName () +
                                 "Email: "+ user.getEmail ());
    }
}
```

Abstraction versus encapsulation

Brighter Blue

Abstraction

- Solves problem at the **design** level
- Used for hiding **unwanted data** and giving **relevant data**
- Focus is on **what** the object does instead on **how** it does it
- **Outer layout** – used in terms of design. e.g. Outer look of a mobile phone – display screen, keypad buttons

Encapsulation

- Solves problem at the **implementation** level
- Used for hiding the **code** and **data** into **single unit** – to protect it from the outside world
- Focus is to **hide** the internal details / mechanics of **how** an object does something
- **Inner layout** – used in terms of implementation. e.g. Inner implementation details of a mobile phone – how the display screen and keypad buttons are connected using circuits

Spot quiz

Brighter Blue

01

Which of the following, enables you to hide, inside the object, both the data fields and the methods that act on that data?

A	Data Abstraction
B	Encapsulation
C	Overloading
D	Control Abstraction

Spot quiz

Brighter Blue

02

Which of the following, plays a pivotal role in parallel programming?

A	Data Abstraction	B	Encapsulation
C	Overloading	D	Control Abstraction

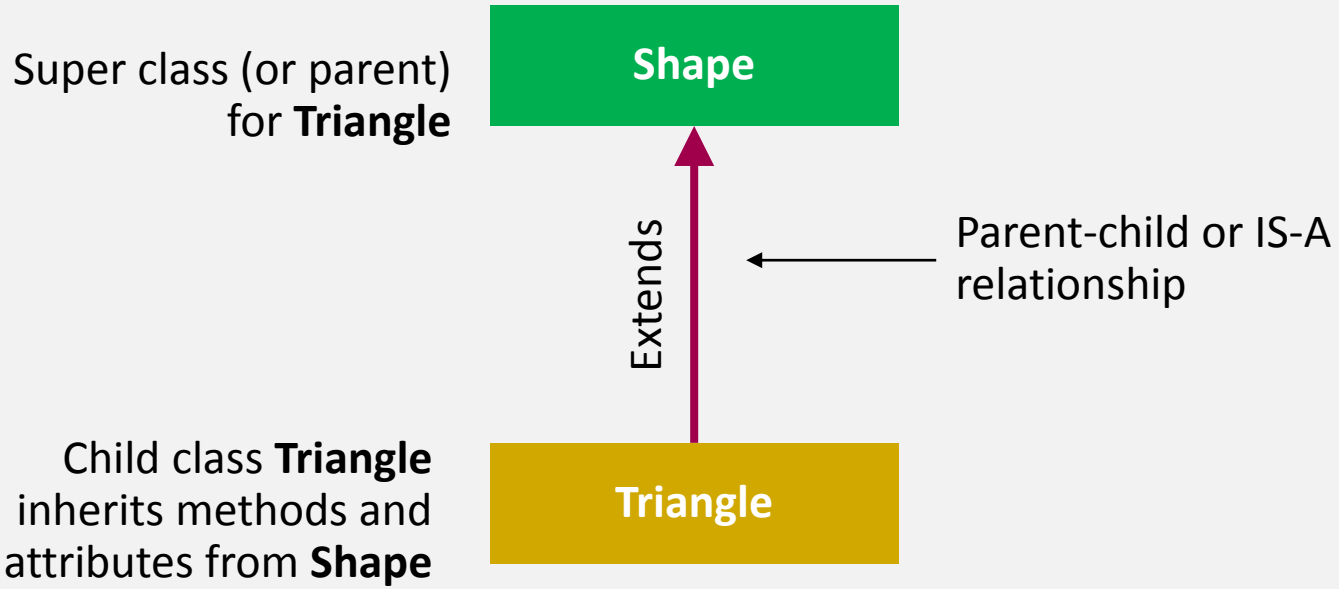
Brighter Blue

02 Inheritance

IBM

What is **inheritance**?

Object-oriented programming (OOP) enables you to define new classes that are based on existing classes—this is known as inheritance.



Spot quiz

Brighter Blue

01

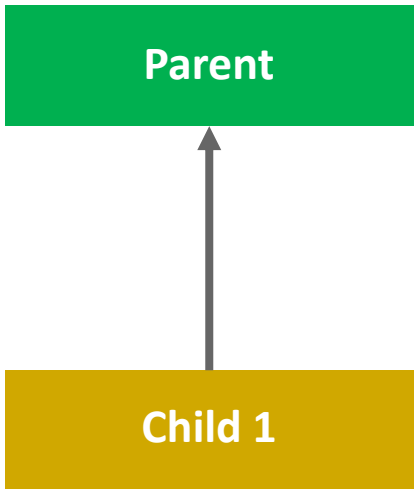
Which of the following are examples of an inheritance hierarchy?

A	Animal extends Dog extends Cat.
B	Cappuccino extends Coffee extends Beverage.
C	Director extends Manager extends Employee.
D	Vehicle extends Car.

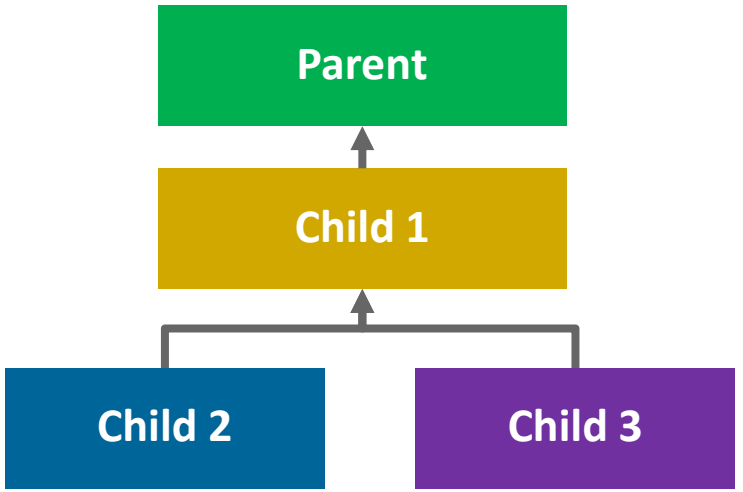
Types of **inheritance**

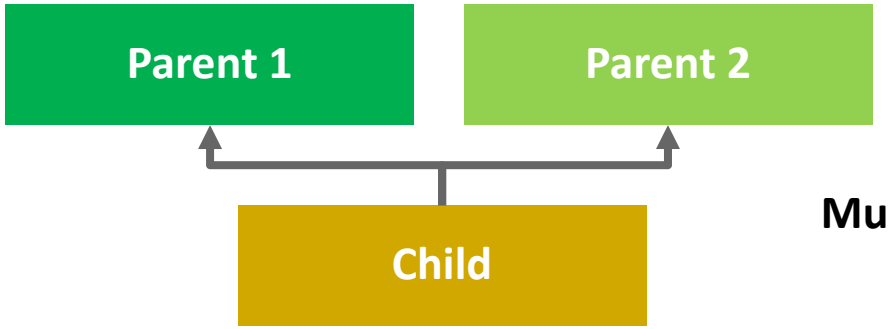
Brighter Blue

Single Inheritance



Multi-level Inheritance



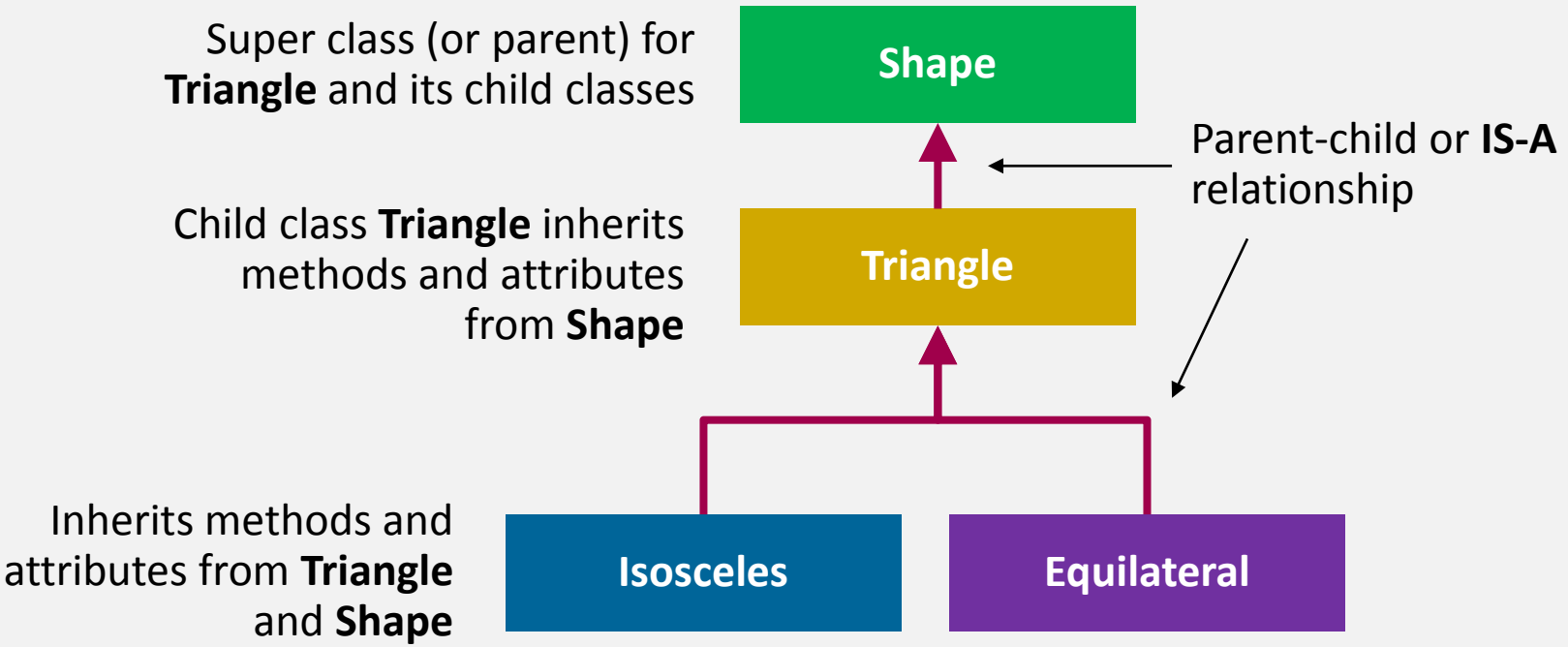


Multiple Inheritance

IBM

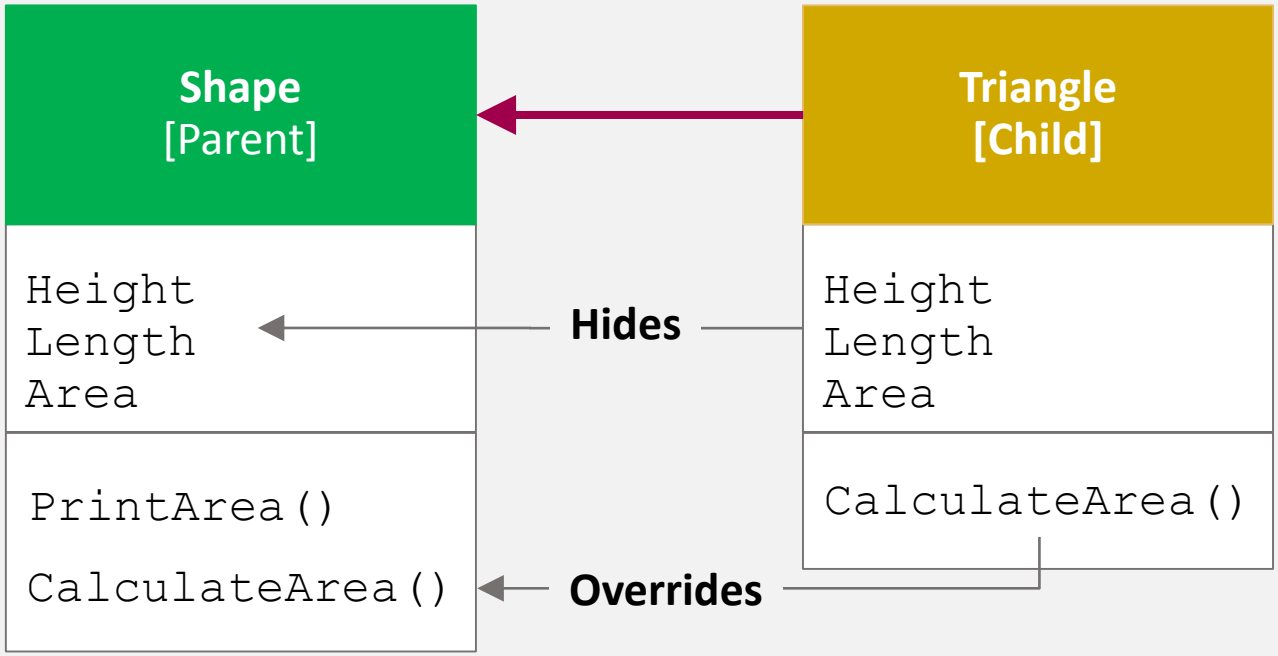
Multi-level inheritance

A sub class can have sub classes of its own.



Overriding inherited methods and variables

Inherited methods can be **overridden** and inherited variables can be **hidden** by creating sub class-specific versions of them.



Implementing inheritance

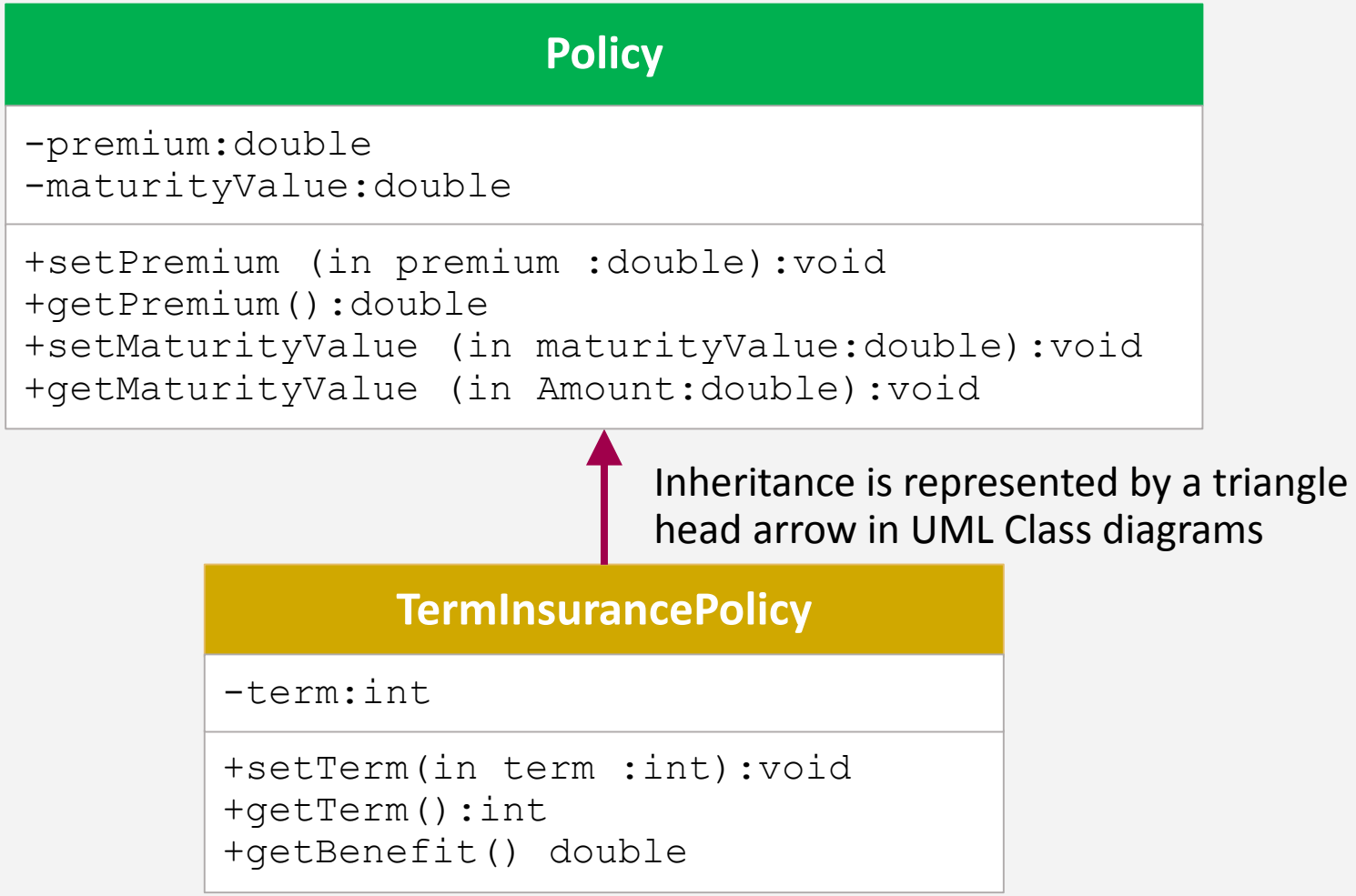
Scenario: Create a class called **TermInsurancePolicy**

Requirements

- Needs all the features of the class **Policy**
- TermInsurancePolicy will have a **pre-defined number of years** before they expire
- Needs to have an extra data called **term** and **relevant methods**

```
public class TermInsurancePolicy extends Policy{

//Subclass-specific Data Members and Methods come here

}
```

Implementing inheritance: class diagram



Implementing inheritance: defining the sub class

Brighter Blue

```
public class TermInsurancePolicy extends Policy{

    private int term;

    public void setTerm(int term){
        this.term = term;
    }
    public int getTerm(){
        return term;
    }
    public double getBenefit(){

        //Calculates benefit based on
        //premium, maturityValue and term values
    }
}
```

Implementing inheritance: **protected access**

The method **getBenefit()** needs to access the data members **premium** and **maturityValue** of the class **Policy**.

maturityValue and **premium** should be declared as **protected** in the Policy class to prevent non-subclasses from accessing the data.

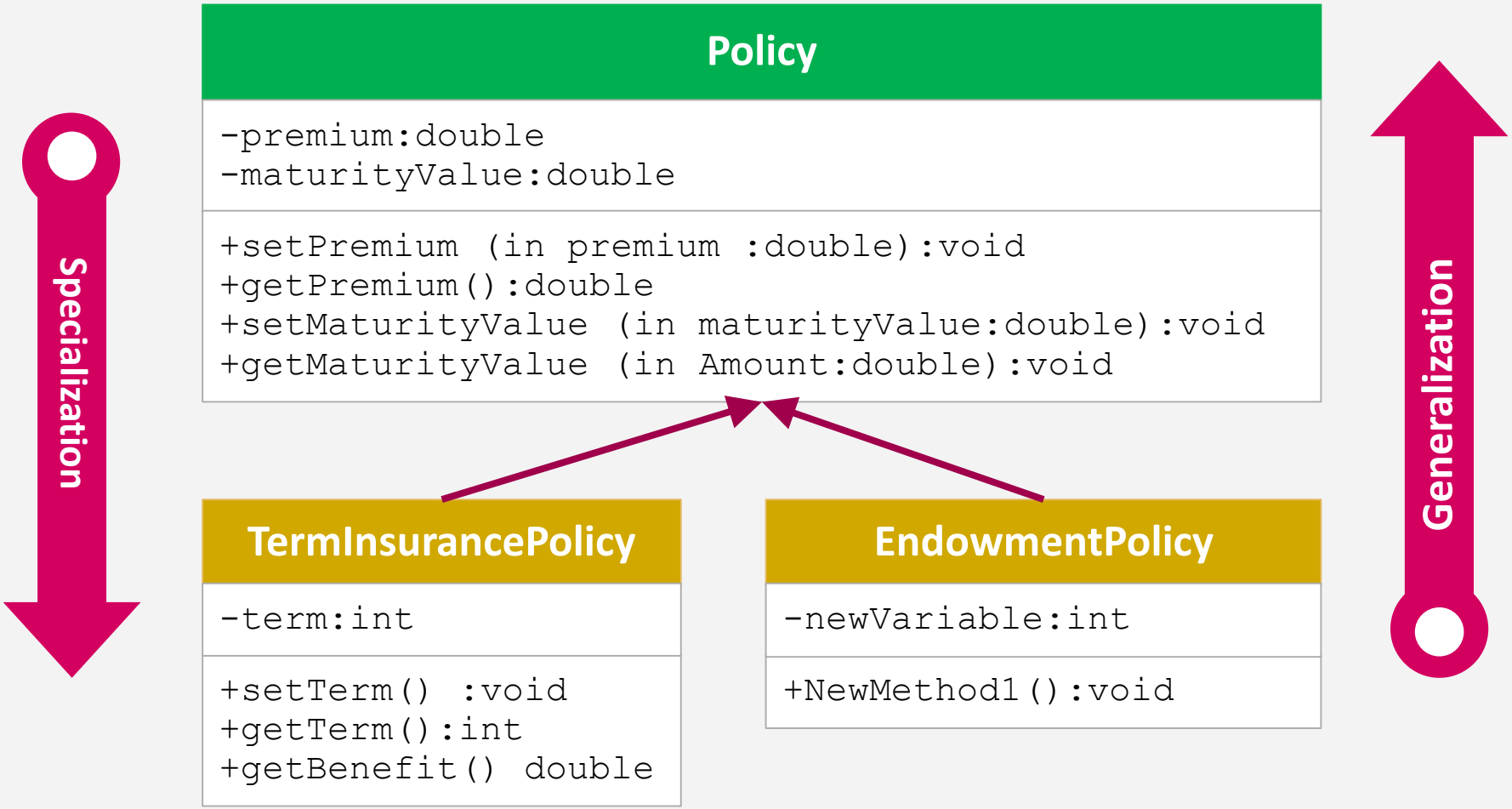
```
public class Policy{

    protected double premium;
    protected double maturityValue;

    //Other Members
}
```

Implementing inheritance: **sub class object**

A **TermInsurancePolicy object** can invoke all the **public methods** of the class Policy and those that are **newly added** in TermInsurancePolicy.

```
TermInsurancePolicy policy = new TermInsurancePolicy();

policy.setPremium(100);
policy.setMaturityValue(5000);
policy.setTerm(36);

System.out.println(policy.getBenefit());
```

Specialization and generalization



Inheritance and **access restrictions**

A sub class **cannot** inherit any of the super class' methods and variables that have **access restrictions**.

These access restrictions include:

Private

The method or variable is not inherited

Default

The method or variable is only inherited if the subclass is defined in the same package as the super class

Overriding an inherited method

```
class Bird {
    String name;
    String color;
    boolean canFly;

    public void move() {
        System.out.println ("I'm on the wing");
        }
    }

class Penguin extends Bird {
    static String canFly = "Penguins can't fly";
    public void move() {
        System.out.println (canFly);
        }
    }
```

Overloading an inherited method

Brighter Blue

You can create methods with the **same name** as an existing method, but **with different arguments**. This is known as overloading a method.

```
class Dog extends Animal {

    public void move() {
    System.out.println ("Dog is moving");
    }

    public void move (int howFar) {
    System.out.println ("Dog's second move method");
    }

    public void move (int howFar, int howFast) {
    System.out.println ("Dog's third move method");
    }
}
```

Constructor chaining

Java ensures that an object is constructed in the correct order, from its base to its sub class.

This is called constructor chaining.

```
class GrandParent {
  int a;
  public GrandParent(int a) {
    this.a = a;
  }
}

class Parent extends GrandParent {
  int b;
  Parent(int a, int b) {
    super(a);
    this.b = b;
  }
}

class Child extends Parent {
  int c;
  Child(int a, int b, int c) {
    super(a, b);
    this.c = c;
  }
  void show() {
    System.out.println("GrandParent's a = " + a);
    System.out.println("Parent's b = " + b);
    System.out.println("Child's c = " + c);
  }
}

public class ConstructorChain {
  public static void main(String[] args) {
    Child object = new Child(7, 8, 9);
    object.show();
  }
}
```

Spot quiz

02

The Furniture class contains a method called “createDeliveryOrder”. Each type of furniture requires specific delivery order. You are creating a Table class.

What is the **best** way to handle the createDeliveryOrder method of the Table class?

A. Implement constructor chaining.

B. Overload the `createDeliveryOrder` method in the Table class.

C. Override the `createDeliveryOrder` method.

Spot quiz

Brighter Blue

03

A Furniture class has two constructors and no explicit no-argument constructor. You derive a Table class from Furniture and define a non-default constructor in Table. The constructor for Table doesn't explicitly call a superclass constructor in Furniture.

What **happens** when you compile Table?

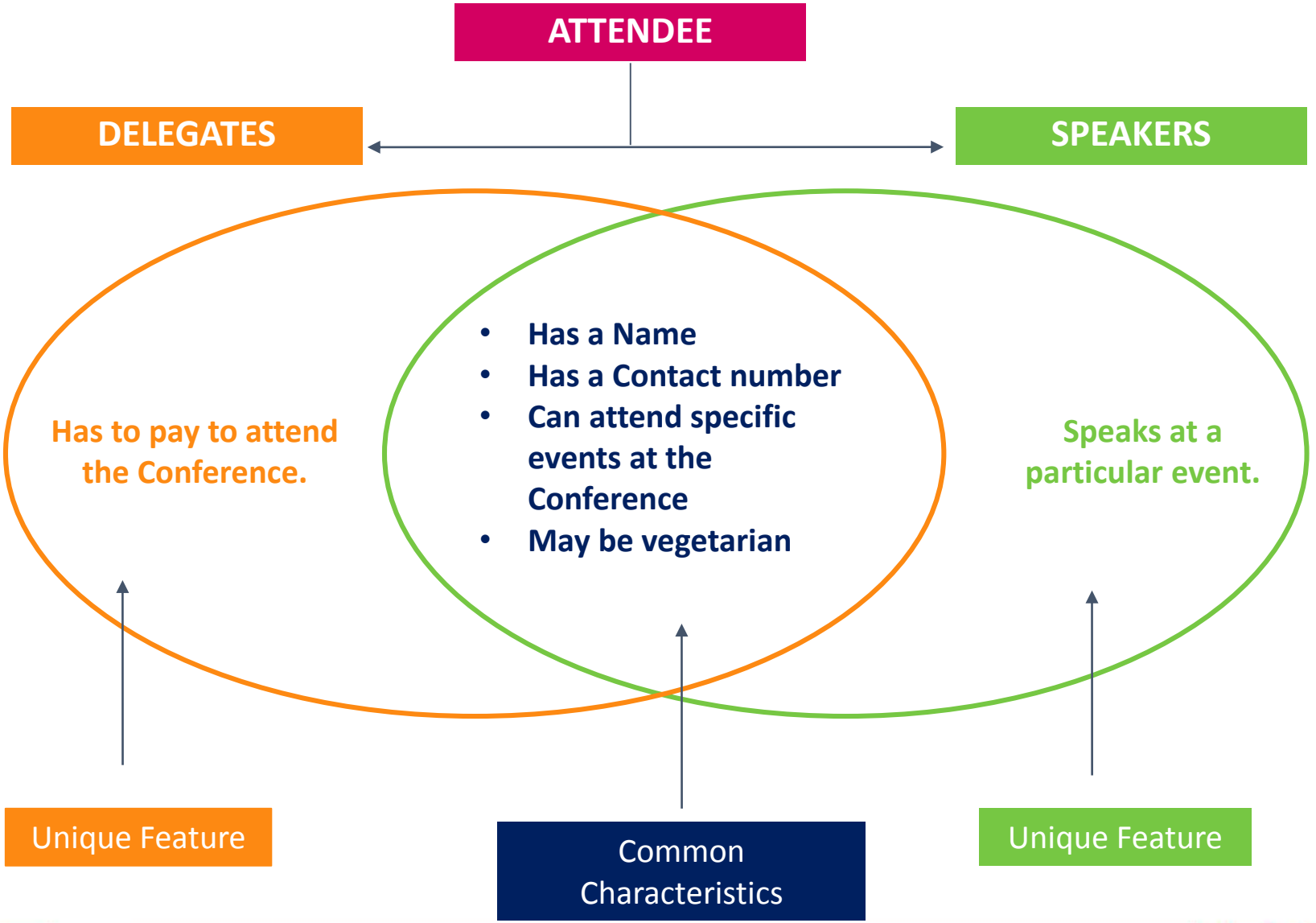
A	Table inherits the constructors from Furniture and an implicit call is made to the matching constructor.	B	The class won't compile.
C	The compiler inserts a call to a constructor in Furniture with the same signature as Table's.	D	The compiler inserts a default, no-argument constructor in both Furniture and Table.

Creation of **sub classes**

Brighter Blue

- A sub class can be created by **including** the **extends** clause and **providing** **overriding** or **implementation** **methods** for abstract codes.
- A sub class inherits **member variables** declared as:
  - **non-private** or **protected**
  - With no **access specifier (default)**
- A sub class doesn't inherit:
  - **private** **member variables** of the superclass
  - Member variables having the **same name as the superclass** – it **hides** them
- A sub class inherits **methods** declared as:
  - **non-private** or **protected**
  - With no **access specifier (default)**
- A sub class doesn't inherit:
  - **private** **methods** of the superclass
  - Methods having the **same name as the superclass** – it **overrides** them

Creating **sub classes**: a walkthrough (1 of 4)



Creating **sub classes**: a walkthrough (2 of 4)

```
/**
* This superclass contains basic attendee info
*
*/
package conference;
public class Attendee {
   //member variables
    String name;
    String company;
    long phoneNumber;
    boolean vegetarian;
    int eventsAttending;
    //constructor
    public Attendee(String name, String
      company, long phoneNumber, boolean vegetarian ){
        this.name = name;
        this.company = company;
        this.phoneNumber = phoneNumber;
        this.vegetarian = vegetarian;
    }
```

Member variables
including personal details

Constructor
initializing personal details

Creating **sub classes**: a walkthrough (3 of 4)

Brighter Blue

The attendee class has 2 methods – **`attendingEvents()`** and **`bookEvent()`**

```
public int attendingEvents() {
    return eventsAttending;
    }
    public boolean bookEvent( ConferenceEvent evt ) {
        if (evt.isBookedOut())
            return false;
        else {
            evt.bookAttendee(this);
            eventsAttending++;
            return true;
        }
    }
```

Creating **sub classes**: a walkthrough (4 of 4)

Brighter Blue

The fully implemented code of the **`class Attendee`** :

```
/**
* This superclass contains basic attendee info
*
*/
package conference;
public class Attendee {
    //member variables
    String name;
    String company;
    long phoneNumber;
    boolean vegetarian;
    int eventsAttending;
    //constructor
    public Attendee(String name, String
        company, long phoneNumber, boolean vegetarian  ){
        this.name = name;
        this.company = company;
        this.phoneNumber = phoneNumber;
        this.vegetarian = vegetarian;
    }
```

Spot quiz

Brighter Blue

04

You must design an Employee database for your company. You have to create classes for Employee, Assistant and Manager.

Which of the following options would you select as the most effective Inheritance Hierarchy, for the 3 classes?

A

A Manager class and an Assistant sub class.

B

An Employee super class and Manager and Assistant sub classes.

C

Assistant and Manager sub classes that override an Employee class.

D

Separate Assistant and Manager classes.

IBM

Creating **sub classes** of a sub class

To code how each Attendee will behave, create a sub class for each **`Attendee`** type.
Use the **`extends`** to create the **`Delegate`** sub class.

```
package conference;
public class  Delegate extends Attendee  {
  // extra variable
  boolean paymentReceived;

  //constructor
  public Delegate (String name, String company,
      long phoneNumber, boolean vegetarian) {
          super (name, company, phone, vegetarian);
          paymentReceived = false;
   }
  //methods
  public boolean bookEvent (ConferenceEvent evt) {
     if (!paymentReceived)
      return false;
    else
      return super.bookEvent(evt);
```

Creating sub classes: using the extends keyword

The **Delegate** class inherits all the package level of the **Attendee** class.

```
package conference;
public class Delegate extends Attendee {
  // extra variable
  boolean paymentReceived;

  //constructor
  public Delegate (String name, String company,
      long phoneNumber, boolean vegetarian ) {
          super (name, company, phone, vegetarian);
          paymentReceived = false;
   }
  //methods
  public boolean bookEvent (ConferenceEvent evt) {
     if (!paymentReceived)
      return false;
    else
      return super.bookEvent(evt);
```

Creating sub classes: declaring a variable

Brighter Blue

Now declare **paymentReceived** variable to check whether **Delegate** has paid the fees.

```
package conference;
public class Delegate extends Attendee {
  // extra variable
  boolean paymentReceived;

  //constructor
  public Delegate (String name, String company,
      long phoneNumber, boolean vegetarian) {
          super (name, company, phone, vegetarian);
          paymentReceived = false;
   }
  //methods
  public boolean bookEvent (ConferenceEvent evt) {
     if (!paymentReceived)
      return false;
    else
      return super.bookEvent(evt);
```

Creating sub classes: using a constructor

`Delegate` class **doesn't inherit** the constructors of **`Attendee`** class, it has to be **invoked** and then **passed on**.

```
package conference;
public class Delegate extends Attendee {
  // extra variable
  boolean paymentReceived;

  //constructor
  public Delegate (String name, String company,
      long phoneNumber, boolean vegetarian) {
          super (name, company, phone, vegetarian);
          paymentReceived = false;
   }
  //methods
  public boolean bookEvent (ConferenceEvent evt) {
     if (!paymentReceived)
      return false;
    else
      return super.bookEvent(evt);
```

Creating sub classes: **invoking** the super class **constructor**

The **payForConference()** sets the value of **paymentReceived = true**

```
public class Delegate extends Attendee {
  // extra variable
  boolean paymentReceived;
  //constructor
  public Delegate (String name, String company, long phoneNumber,
  boolean vegetarian) {
          super (name, company, phone, vegetarian);
          paymentReceived = false;
   }
  //methods
  public boolean bookEvent (ConferenceEvent evt) {
     if (!paymentReceived)
      return false;
    else
      return super.bookEvent(evt);
  }
   public void payForConference() {
   paymentReceived = true;
  }
}
```

Creating sub classes: **invoking method** of the super class

The **`super.bookEvent(evt)`** is called to invoke the method of the **`Attendee`** class.

```
public class Delegate extends Attendee {
  // extra variable
  boolean paymentReceived;
  //constructor
  public Delegate (String name, String company, long phoneNumber,
  boolean vegetarian) {
          super (name, company, phone, vegetarian);
          paymentReceived = false;
   }
  //methods
  public boolean bookEvent (ConferenceEvent evt)  {
     if (!paymentReceived)
      return false;
    else
      return super.bookEvent(evt);
  }
   public void payForConference () {
    paymentReceived = true;
  }
}
```

Creating sub classes: **exclusive method** of the sub class

Some conditions apply only to the **`Delegate`** class.

```
public class Delegate extends Attendee {
  // extra variable
  boolean paymentReceived;
  //constructor
  public Delegate (String name, String company, long phoneNumber,
  boolean vegetarian) {
          super (name, company, phone, vegetarian);
          paymentReceived = false;
   }
  //methods
  public boolean bookEvent (ConferenceEvent evt) {
     if (!paymentReceived)
      return false;
    else
      return super.bookEvent(evt);
  }
   public void payForConference () {
     paymentReceived = true;
  }
}
```

Creating sub classes: **overriding** the super class **method**

The **`bookEvent()`** for the **`Delegate`** class is now a **modified version** and **overrides** that of the **`Attendee`** version.

```
public class Delegate extends Attendee {
  // extra variable
  boolean paymentReceived;
  //constructor
  public Delegate (String name, String company, long phoneNumber,
  boolean vegetarian) {
          super (name, company, phone, vegetarian);
          paymentReceived = false;
   }
  //methods
  public boolean bookEvent (ConferenceEvent evt) {
     if (!paymentReceived)
      return false;
    else
      return super.bookEvent(evt);
  }
   public void payForConference () {
    paymentReceived = true;
  }
}
```

Spot quiz

Brighter Blue

05

Which methods can access to private attributes of a class?

A	Only static methods of the same class.
B	Only instance of the same class.
C	Only methods those defined in the same class.
D	Only classes available in the same package.

Spot quiz

Brighter Blue

06

Given a class named Employee, which of the following is a valid constructor declaration for the class?

A

```
Employee(Employee e)
{}
```

B

```
Employee Employee()
{}
```

C

```
Private final
Employee (){}
```

D

```
Static void Employee
(){}
```


IBM

Brighter Blue

03 Polymorphism

IBM

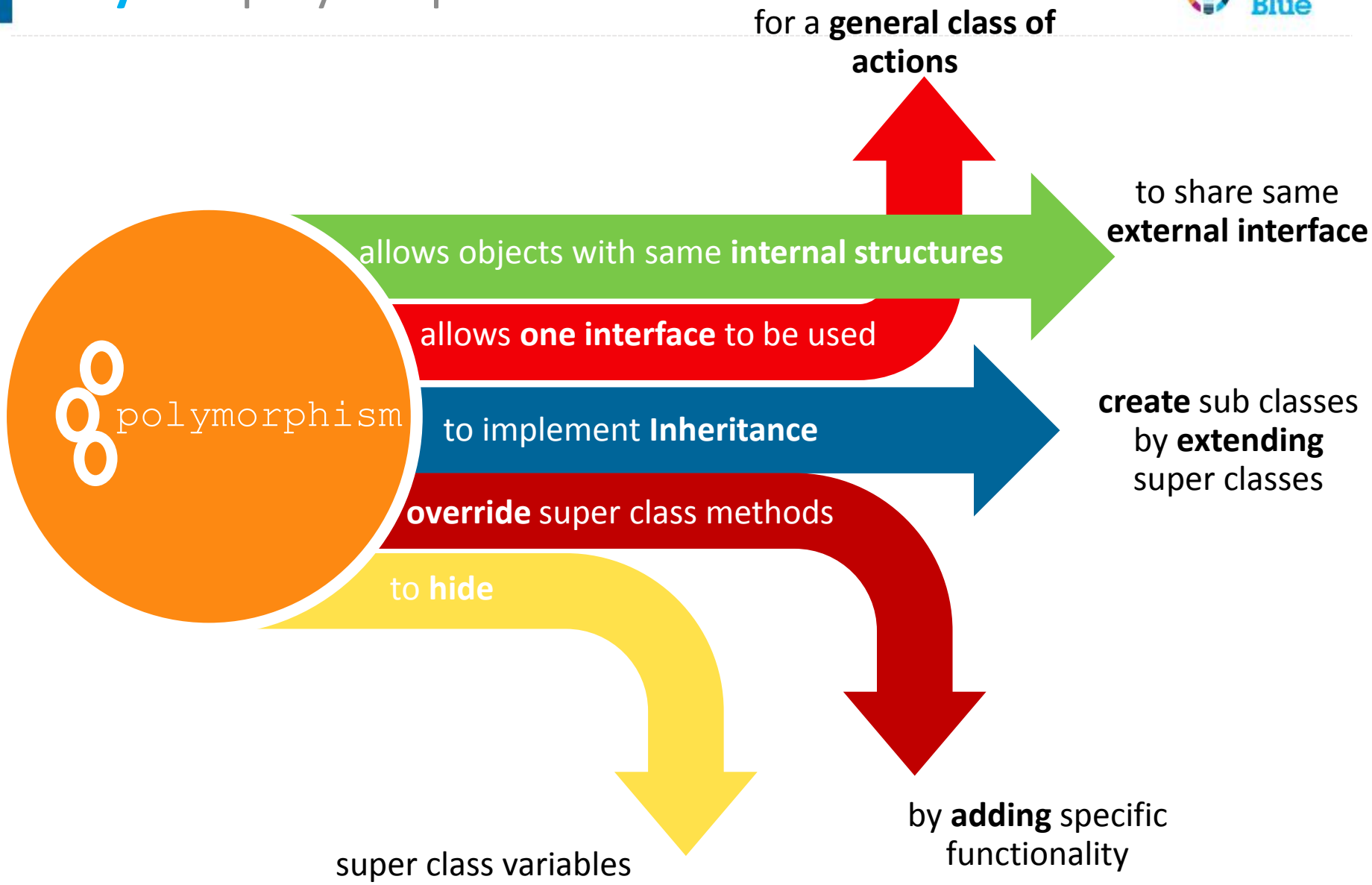
What is **polymorphism**?

Is a concept by which a **single action** can be performed in **various ways**

A superclass method can **adopt different forms**, depending on the subclass

Ability of an object to take **many forms**

Why use polymorphism?



Activity

Brighter Blue

Polymorphism demo

IBM

Spot quiz

01

You have an abstract Vehicle class. The Ford class extends the Car class, which extends the Vehicle class, and overrides its getSpec method. You assign a Ford reference to a variable , vehicle or type of vehicle.

Which **getSpec** method is called through vehicle?

```
Vehicle a = new Ford();

a.getSpec();
```

A The `getSpec` method of the **Car object**

B The `getSpec` method of the **Ford object**

C The `getSpec` method of the **Vehicle object**

What is **binding**?

An **association** of the **Method Definition** to the **Method Call**

2 types

Dynamic/Late Binding

- Can't be resolved at compile time by compiler
- Perfect example – Overriding

Static / Early Binding

- Can be resolved at compile time by compiler
- Perfect example – Overloading

Static binding versus Dynamic binding

Brighter Blue

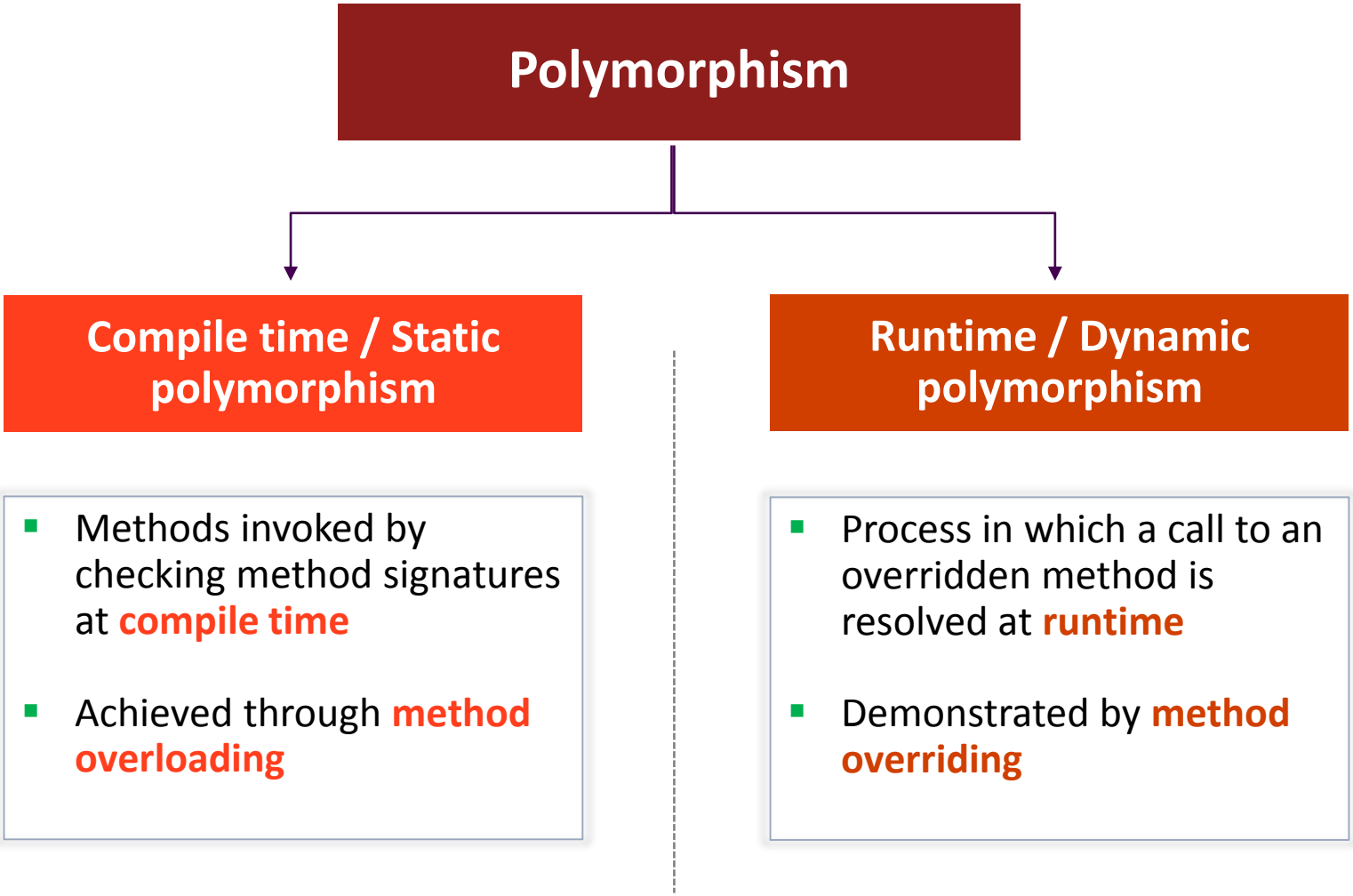
Static Binding

- Happens at **compile time**
- Binding of **private, static** and **final methods** at compile time
- Java uses it for **overloaded methods**

Dynamic Binding

- Happens at **run time**
- Binding of **overridden methods** at run time
- Java uses it for **overridden methods**

Types of **polymorphism**



Understanding dynamic polymorphism

Brighter Blue

Note the `read()` and the `write()` of the `public class Policy`

```
public class Policy{
        //Data Members and Other Methods
        public void read() {
                //Read the premium, maturityValue
                //and other data
        }
        public void write() {
                System.out.println("Premium:" + premium);
                System.out.println("Maturity Value:" +
                                        maturityValue);
                //Write other data
        }
}
```

Understanding **dynamic polymorphism** (continued)

Brighter Blue

Class `TermInsurancePolicy` needs similar methods for reading and writing its data members

```
public class TermInsurancePolicy extends Policy{
	private int term;
	public void read(){
		//Read term
	}
	public void write(){
		System.out.println(term);
	}
	//Other Methods
}
```

Understanding **method overriding**

The class definition of `TermInsurancePolicy` extends `Policy` class

```
public class TermInsurancePolicy extends Policy{
        private int term;
        public void read()  {
                super.read()  ;
                //Read term
        }
        public void write() {
                super.write() ;
                System.out.println(term);
        }
        //Other Methods
}
```

Understanding **dynamic binding** : 1 of 3

Brighter Blue

A reference to a super class can refer to a sub class object

```
public class EndowmentPolicy extends Policy{
	//Data Members and Method
	public void read(){
		super.read();
		//Read other data members
	}
	public void write(){
		super.write();
		//Write other data members
	}
}
```

Understanding **dynamic binding** : 2 of 3

The `TermIsurancePolicy` or the `EndowmentPolicy` has been chosen at runtime

```
char choice;
//Read choice, T for TermInsurancePolicy, E for EndowmentPolicy
if (choice == 'T'){
     TermInsurancePolicy tiPolicy = new TermInsurancePolicy();
     tiPolicy.read();
     tiPolicy.write();
}
else{
     EndowmentPolicy ePolicy = new EndowmentPolicy();
     ePolicy.read();
     ePolicy.write();
}
```

Understanding **dynamic binding** : 3 of 3

Brighter Blue

```
char choice;
Policy policy;
if (choice == 'T'){
     policy = new TermInsurancePolicy();
}
else{
     policy = new EndowmentPolicy();
}
policy.read();
//Calls the read of TermInsurancePolicy or EndowmentPolicy
//Decided at runtime
policy.write();
//Calls the write of TermInsurancePolicy or EndowmentPolicy
//Decided at runtime
```

Understanding runtime polymorphism

The same command `policy.read` is used to read `TermInsurancePolicy` or `EndowmentPolicy`

```
char choice;
Policy policy;
if (choice == 'T'){
    policy = new TermInsurancePolicy();
}
else{
    policy = new EndowmentPolicy();
}
policy.read();
//Calls the read of TermInsurancePolicy or EndowmentPolicy
//Decided at runtime
policy.write();
//Calls the write of TermInsurancePolicy or EndowmentPolicy
//Decided at runtime
```

Static polymorphism versus Dynamic polymorphism

Brighter Blue

Static Polymorphism

- Function overloading
- Resolved during compilation time

Dynamic Polymorphism

- Function overriding
- Resolved during run time

Spot quiz

Brighter Blue

02

What is Binding?

A	It is to provide access to an object only through its member functions, while keeping the details private.
B	It is an association of the method definition to the method call.
C	It is a concept by which a single action can be performed in various ways.
D	It is a special method that has the same name as the class and is invoked automatically whenever an object of the class is instantiated.

Spot quiz

Brighter Blue

03 What is Polymorphism?

A	It is when a single super class has many sub classes**.**
B	It is when a single variable is used with several different types of related objects at different places in a program.
C	It is when a class has several methods with the same name but different parameter types.
D	It is when a program uses several different types of objects, each with its own variable.

Brighter Blue

Day 02 Practice Exercises:

1. Abstraction
2. Polymorphism (Part 1 and 2)

Refer the Additional Exercises hand out