Inheritance, Abstraction, and Polymorphism

Java Developer

StayAhead Training; January, 2023

Inheritance

Inheritance, Abstraction, and Polymorphism

- Inheritance is the inheriting of state and behaviour by one class from another

```
class LibraryBook extends Book {}
```

- Imagine that Book's fields and methods are copied into LibraryBook
- Inheritance describes an **is-a** relationship, e.g. a LibraryBook is-a Book
- The inherited class is the super/base/parent class
- The inheriting class is the sub/derived/child class

The Purpose of Inheritance

Inheritance, Abstraction, and Polymorphism

- In part inheritance serves to reduce code duplication
- Where two or more classes share common state & behaviour then inheritance can reduce/eliminate that duplication
- Suppose you have AudioTrack and Video classes that each have name, author, and duration fields then you might move those common fields into a superclass named Media
- BUT, inheritance is not just a tool to reduce code duplication…

The Purpose of Inheritance

Inheritance, Abstraction, and Polymorphism

- Inheritance is the means by which we can exploit polymorphism (soon)
- The inheritance relationship **must** make sense in plain English, e.g.

```
// a Team is-a Player? NO
class Team extends Player {}

// a Defender is-a Player? YES
class Defender extends Player {}
```

Single vs. Multiple Inheritance

Inheritance, Abstraction, and Polymorphism

- Single inheritance: a class can inherit from one class only
- Multiple inheritance: a class can inherit from many classes
- Java permits **single inheritance only**
- You and I have multiple parents, so why not Java classes?
- What if both parents have a method named play that accepts no args?
- Some languages have complex rules enabling multiple inheritance; not Java

What is Inherited?

Inheritance, Abstraction, and Polymorphism

- All fields and methods, both static and instance, are inherited or not according to the assigned access modifier

- Constructors are **not** inherited

- The name of the subclass constructor must differ from that of the superclass

- Recall that the constructor name must match the class name

Extending a Super Class

Inheritance, Abstraction, and Polymorphism

- Use the extends keyword to inherit from another class, e.g.

```
class Directory extends File {}
```

- The word, extends, is instructive - to extend yourself is to do more than you have done previously
- In this example, a Directory is-a File that has more and/or does more
- If the subclass doesn't have additional state and/or behaviour
  then there's no point to it

Shadowing

Inheritance, Abstraction, and Polymorphism

- Shadowing is the hiding of an inherited field, e.g.

```
class A {
  int x = 1;
}

class B extends A {
  // the subclass's x shadows the super class's x
  int x = 2;
}
```

- The superclass's field is still there and can be accessed in the subclass

Overriding

Inheritance, Abstraction, and Polymorphism

- Overriding is the implementation of a superclass method in the subclass, e.g.

```
class A {
  void greet() { System.out.println("Hello"); }
}

class B extends A {
  void greet() { System.out.println("G'day"); }
}
```

- The superclass's method is still there and can be accessed in the subclass

Overriding

Inheritance, Abstraction, and Polymorphism

- An annotation is metadata - it provides additional info about your code
- Some annotations exist during compilation only, some persist during runtime
- The `@Override` annotation marks a subclass method as overriding a superclass method
- It, like all method annotations, is typically placed on the line immediately above the method declaration
- On encountering the `@Override` annotation the compiler checks that the method does in fact override a superclass method

Overriding

Inheritance, Abstraction, and Polymorphism

```
class A {
  void greet() { System.out.println("Hello"); }
}

class B extends A {
  @Override // OK
  void greet() { System.out.println("G'day"); }
}

class C extends A {
  @Override // compilation error - overload, not override
  void greet(String name) { System.out.println("Hi " + name); }
}
```

Subclass Constructors

Inheritance, Abstraction, and Polymorphism

- Each subclass constructor must call a superclass constructor
- The constructor is (or should be) responsible for ensuring that all the fields are initialised and we don't wind up with invalid objects
- We're forced to call a superclass constructor to ensure that all the inherited fields are initialised properly
- How does one call the superclass constructor…

The super Keyword

Inheritance, Abstraction, and Polymorphism

- The super keyword is used to access superclass members, e.g.

```
class B extends A {
  void greet() {
    super.greet();
    System.out.println("Nice to meet you");
  }
}
```

- The super keyword can only be used within a subclass

The super Keyword

Inheritance, Abstraction, and Polymorphism

- The super keyword can be used to call the superclass constructor, e.g.

```
class B extends A {
  B() {
    super();
    // B-specific initialisation
  }
}
```

- The compiler will add the call to the superclass constructor if you don't

The super Keyword

Inheritance, Abstraction, and Polymorphism

- Some devs believe that calling the superclass constructor results in the instantiation of the superclass

- In an inheritance hierarchy where, for example,
  E extends D extends C extends B extends A…
  then instantiating E would result in the creation of five objects!

- This view of inheritance is **WRONG!**

- Calling the superclass constructor ensures the inherited fields are initialised - it does **not** result in the creation of a superclass object

Inheritance Polymorphism

Inheritance, Abstraction, and Polymorphism

- Polymorphism means many forms (one object; many types)
- It is a by-product of inheritance
- If B extends A then an instance of B is also an instance of A
- That is, an instance of B is guaranteed to have all the state and behaviour that an instance of A does (it must)
- One object (instance of B); many types (A and B)

Inheritance Polymorphism

Inheritance, Abstraction, and Polymorphism

- Consider the following:

```
Book book = new LibraryBook();
```

- The variable type is the superclass and the object type the subclass
- This works because a LibraryBook **is-a** Book
- There is nothing in the Book class that a LibraryBook object doesn't have
- However, a LibraryBook object may have fields and/or methods that are not defined in the Book class

The Polymorphism Rule*

Inheritance, Abstraction, and Polymorphism

The variable type dictates **WHAT** you can do

The object type dictates **HOW** you do it

The Polymorphism Rule*

Inheritance, Abstraction, and Polymorphism

```java
class A {
  void greet() { System.out.println("Hello"); }
}

class B extends A {
  void greet() { System.out.println("G'day"); }
}

A a = new B();

a.greet(); // G'day
```

The Polymorphism Rule*

Inheritance, Abstraction, and Polymorphism

- In the example on the previous slide, `a.greet()` yields G'day because the object type is B, therefore it is the version of greet in class B that is called
- The variable type is A, so only the methods defined in class A may be called
- If class B were to include additional methods (beyond greet) then those methods could not be called using a variable of type A

Upcasting and Downcasting

Inheritance, Abstraction, and Polymorphism

- Upcasting is the assignment of a subclass variable to a superclass variable

```
LibraryBook libraryBook = new LibraryBook();

Book book = libraryBook;
```

- This type of casting is performed automatically and without having to enclose the cast-to type in parentheses
- Every LibraryBook **is-a** Book

Upcasting and Downcasting

Inheritance, Abstraction, and Polymorphism

- Downcasting is the assignment of a superclass variable to a subclass variable

```
Book book = new LibraryBook();

LibraryBook libraryBook = (LibraryBook) book;
```

- This type of casting must be performed manually by enclosing the cast-to type in parentheses
- Not every Book **is-a** LibraryBook
- Casting deals only with the variable type, not the object type

The instanceof Operator

Inheritance, Abstraction, and Polymorphism

- The instanceof operator is used to test the type of an object, e.g.

```
Book book = new LibraryBook();

assert book instanceof Book;

assert book instanceof LibraryBook;
```

- In this case both assertions are true - the object referenced by book is both Book and LibraryBook (that's polymorphism!)

The instanceof Operator

Inheritance, Abstraction, and Polymorphism

- An instanceof test is necessarily accompanied by a downcast, e.g.

```
if (book instanceof LibraryBook) {
  var libraryBook = (LibraryBook) book;
  // some LibraryBook specific operation
}
```

- Since Java 12 this can be simplified as follows, e.g.

```
if (book instanceof LibraryBook libraryBook) {
  // some LibraryBook specific operation
}
```

The Benefits of Polymorphism*

Inheritance, Abstraction, and Polymorphism

- Let's assume *n* clients, some of which prefer to be contacted by phone, some by text message, and some by email
- We might do…

```
class Client {
  void sendMessage(String message) {
    if (contactPreference.equals("phone")) {
      // TODO
    } else if (contactPreference.equals("text")) {
      // TODO etc.
```

The Benefits of Polymorphism*

Inheritance, Abstraction, and Polymorphism

- Code that is executed conditionally based on the value of some field is, generally speaking, bad practice

- If the means of communication change in the future then the class's code will have to be changed too (we should avoid making changes to working code)

- Inheritance and polymorphism offer a solution…

The Benefits of Polymorphism*

Inheritance, Abstraction, and Polymorphism

```
class Client {
  void sendMessage(String message) {
    // TODO
  }
}

class ContactByPhoneClient extends Client { ... }

class ContactByTextClient extends Client { ... }

class ContactByEmailClient extends Client { ... }
```

The Benefits of Polymorphism*

Inheritance, Abstraction, and Polymorphism

- Each subclass of Client overrides the sendMessage method with its own specific implementation
- If the means of communication change in the future then we need only add/remove Client subclasses; we needn't change working code
- When dealing with many Clients, we can code to the superclass and effectively ignore (be ignorant of) the various sub-types

The Benefits of Polymorphism*

Inheritance, Abstraction, and Polymorphism

```
// the variable type dictates WHAT we can do
var clients = new ArrayList<Client>();

clients.add(new ContactByTextClient());
clients.add(new ContactByPhoneClient());
clients.add(new ContactByEmailClient());

for (var client : clients) {
  // the object type dictates HOW the message is sent
  client.sendMessage("Hello world");
}
```

final Classes and Methods

Inheritance, Abstraction, and Polymorphism

- Recall that a final variable/field is one that cannot be reassigned
- A final method is one that cannot be overridden in the subclass, e.g.

```
final void cantBeOverridden() { ... }
```

- A final class is one that cannot be subclassed, e.g.

```
final class CantBeSubclassed { ... }
```

Sealed Classes (since Java 15)

Inheritance, Abstraction, and Polymorphism

- A sealed class is one that must be inherited from and which must specify its subclasses (it permits only a limited set of subclasses), e.g.

```
sealed class Account permits SavingsAccount, ISA {}
```

- The `SavingsAccount` and `ISA` classes must themselves be either:
  - `final`: cannot be subclassed
  - `sealed`: must specify its subclasses
  - `non-sealed`: can be subclassed

```
non-sealed class SavingsAccount {}
```

The Object Class

Inheritance, Abstraction, and Polymorphism

- The java.lang.Object class was/is the very first Java class

- Every class ultimately inherits from Object

- Unless specified otherwise, each of your classes inherits from Object, e.g.

```
class Book {} // is effectively the same as...
class Book extends Object {}
```

- The Object class has no fields but does have some methods

The Object Class

Inheritance, Abstraction, and Polymorphism

- The **hashCode** method returns an numeric representation of the object and is used in hashing collections, like HashMap
- The **equals** method accepts another Object and returns true if this and the other Object are equal
- NB: the default implementation of equals uses the equality operator and so tests for equality of reference by default
- The hashCode and equals methods should always be overridden together

The Object Class

Inheritance, Abstraction, and Polymorphism

- The **toString** method returns a String representation of the object which, by default, is a hexadecimal representation of the object's hash code

- The toString method should be overridden to return a "concise but informative representation that is easy for a person to read" - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#toString()

- The specifics of overriding hashCode and equals is beyond the scope of this course but your editor includes tools to help you do so

Abstractions and Implementations

Inheritance, Abstraction, and Polymorphism

- In Java an abstraction is a class that **cannot** be instantiated; it is too generalised/not specific enough
- It is a superclass that defers to its subclasses to provide the specifics
- To use the previous example, the Client superclass defers to its subclasses to specify how to send messages (though Client is not abstract)
- An implementation is a subclass of an abstract superclass/a class that provides the implementation details - it makes the abstract concrete

The Purpose of Abstraction

Inheritance, Abstraction, and Polymorphism

- In regular inheritance:
  - the subclass may override inherited methods
  - the superclass must provide default implementations
- Where the superclass is an abstraction:
  - the subclass must override abstract methods
  - the super class need not provide default implementations

Abstract Classes

Inheritance, Abstraction, and Polymorphism

- An abstract class is one that cannot be instantiated - it must be subclassed
- The abstract keyword is used to mark a class as abstract, e.g.

```
abstract class Client { ... }

var client = new Client(); // compilation error
```

- An abstract class may have fields, concrete (non-abstract) methods, and abstract methods
- A class with one or more abstract methods must be abstract

Abstract Methods

Inheritance, Abstraction, and Polymorphism

- An abstract method is one that must be overridden in the subclass(es)
- The abstract keyword is used to mark a method as abstract
- An abstract method has no method body (no braces), e.g.

```
abstract void sendMessage(String message);
```

- An abstract method specifies **WHAT** to do, but **not HOW** to do it

Extending an Abstract Class

Inheritance, Abstraction, and Polymorphism

- Inheriting from an abstract class takes the usual form, e.g.

```
class MyImplementation extends MyAbstraction {}
```

- When you inherit from an abstract class the compiler will force you to override all of the inherited abstract methods (unless the subclass is abstract, too)
- Inheriting from an abstract class means inheriting obligations

Interfaces

Inheritance, Abstraction, and Polymorphism

- An <u>interface</u> is an abstract class that has **abstract methods only**
- The interface keyword is used to define an interface, e.g.

```
interface List { ... }

var list = new List(); // compilation error
```

- An interface **cannot** have fields or concrete methods (there are some exceptions)

Interfaces

Inheritance, Abstraction, and Polymorphism

- Interface methods are public and abstract by default, e.g.

```
interface List {
  // the modifiers are redundant
  public abstract void add(Object o);
}

interface List {
  void add(Object o);
}
```

Implementing an Interface

Inheritance, Abstraction, and Polymorphism

- The implements keyword is used to implement (inherit from) an interface, e.g.

```
class MyList implements List { ... }
```

- Unlike regular inheritance, a class can implement many interfaces, e.g.

```
class C implements D, E, F { ... }
```

- A class can inherit from a superclass & implement one or more interfaces, e.g.

```
class B extends A implements D { ... }
```

Implementing an Interface

Inheritance, Abstraction, and Polymorphism

```
interface A {
  void go();
}

interface B {
  void go();
}

// is this OK?
class C implements A, B { … }
```

Implementing an Interface

Inheritance, Abstraction, and Polymorphism

```
interface A {
  void go();
}

interface B {
  void go();
}

// is this OK? YES - C must override go; it doesn't matter
// how many interfaces impose the obligation
class C implements A, B { … }
```

Implementing an Interface

Inheritance, Abstraction, and Polymorphism

```
class A {
  void go() { System.out.println("Going..."); }
}

interface B {
  void go();
}

// is this OK?
class C extends A implements B { … }
```

Implementing an Interface

Inheritance, Abstraction, and Polymorphism

```
class A {
  void go() { System.out.println("Going..."); }
}

interface B {
  void go();
}

// is this OK? YES - B imposes the obligation; A provides
// the implementation; there's nothing for C to do
class C extends A implements B { … }
```

Interface Polymorphism

Inheritance, Abstraction, and Polymorphism

- Polymorphism means many forms (one object; many types)
- It is a by-product of inheritance
  (to implement an interface is to inherit from an abstract class)
- If B implements A then an instance of B is also an instance of A
- That is, an instance of B is guaranteed to have all the behaviours that are defined by A
- One object (instance of B); many types (A and B)

Interface Polymorphism

Inheritance, Abstraction, and Polymorphism

- Consider the following:

```
List list = new MyList();
```

- The variable type is the interface and the object type the implementing class
- This works because MyList **is-a** List
- There is nothing in the List interface that a MyList object doesn't have
- However, a MyList object may have fields and/or methods that are not defined in the List interface

Abstract Class vs. Interface

Inheritance, Abstraction, and Polymorphism

- Suppose you conclude that Client ought to be an abstraction
- Do you make it an abstract class or an interface?
- **If it has fields and concrete methods it should be an abstract class**
- **If it has abstract methods only it should be an interface**
- It is very likely that Client would be an abstract class

Interface Default Methods

Inheritance, Abstraction, and Polymorphism

- Since Java 8 an interface can have default concrete methods, e.g.

```
interface A {
  default void go() {
    // TODO
  }
}
```

- The implementing class may or may not override the default method
- Implementing two or more interfaces with duplicate default methods produces a compilation error

Interface Default Methods

Inheritance, Abstraction, and Polymorphism

- Interface default methods were introduced to enable the addition of functionality to existing interfaces without breaking code the world over
- They should **not** form part of your app design
- In fact, we think it best to steer clear of them all together

Interface Static Methods

Inheritance, Abstraction, and Polymorphism

- Since Java 8 an interface can have static concrete methods, e.g.

```java
interface A {
  static void go() {
    // TODO
  }
}
```

- Note that the static method is **not** inherited by the implementing class

Interface private Methods

Inheritance, Abstraction, and Polymorphism

- Since Java 8 an interface can have private concrete methods, e.g.

```
interface A {
  private void go() {
    // TODO
  }
}
```

- An interface private method can only be called by default methods in the given interface