# Subsection 1: Explain inheritance and its benefits

When you design a class you can choose to include functionality from another class by extending that class. Your new class will include the methods and fields from the class it extends plus any new methods and fields that you add yourself. Often the reason for extending a class with a new class like this is to specialise on the behaviour of the extended class. This means that the class being extended has general functionality that could be shared by a variety of other specialised classes. Let’s look at an example.

When explaining inheritance it is common to use the analogy of animals and their classifications. Let’s take the Cat Family as our example. Cat is a family of animals that can be divided into smaller groups knows as genus (Felis, Panthera, [Puma](https://a-z-animals.com/animals/puma/)) and each genus has a species (domestic cat, tiger, leopard, and cougar.

At hierarchical structure can be drawn to represent these relationships.

At the top of the hierarchy is the **Cat**. This can be seen as an abstract concept of the cat family, because no actual cat exists, only specialisations of the cat concept in terms of the **Domestic** **Cat**, **Leopard** and **Cougar**. Between the Family and the Species is the Genus. This groups the species into categories of similar but distinct behaviour. So, for the purpose of this example, I could say that the characteristic that puts species into the **Panthera** genus is that they have a nocturnal lifestyle, those that fit into the **Puma** genus have a twilight lifestyle and those that fit into the **Felis** genus have a day-time lifestyle. But still, there is no actual animal that is just a **Panthera**. The Genus is still an abstract idea about a cat, but a slight more specialised one than the **Cat** concept.

At the species level we start to talk about the concrete animal itself and about specific details such as its markings, size, speed, and the food it likes to eat.

## Java models the real world

Java attempts to model the real world and represent it with objects. Relationships between objects can also be modelled. We have seen throughout this course various examples of classes relating to one another. Now let’s complete our knowledge by looking in to the concept of inheritance.

A representation of the hierarchical chart’s structure can be modelled with Java code using interfaces, abstract classes and subclassing.

### An interface

An interface is a special type of class that represents the most abstract concept in a hierarchical relationship. It is constructed with the interface keyword instead of the class keyword. The code snippet below shows an example of this.

**interface Cat {}**

I have represented the **Cat** concept as an interface because it is the most abstract idea in our model and no actual **Cat** can exist, as it is only a concept.

Representing the **Cat** concept as an interface is the right choice because the rules of Java do not allow an instance of an interface to exist. So no **Cat** object can be created and so exists just as a concept in our model.

If I attempted to create an instance of the **Cat** interface with the new keyword the code would not compile.

**Cat cat = new Cat(); // will not compile**

So you might ask what the purpose of the interface is if you cannot make an object from it. Well it exists in order to represent a type (category/family) of objects; in this case, it represents all the types of cats in the **Cat** hierarchy.

### Implementing interfaces

Interfaces are implemented with the **implement** keyword and multiple interfaces can be implemented. The following code snippet is perfectly valid.

public abstract class Panthera implements Cat, Mammal, Animal

Interfaces can be implemented by any class but not by interfaces. The following code snippet shows that the **Tiger** class can extend the **Panthera** abstract class and implement the **Animal** interface.

public class Tiger extends Panthera implements Animal

Interface cannot implement other interfaces however they can extend multiple other interfaces as shown in this code snippet where the **Cat** interface extends the **Mammal** and **Animal** interfaces.

public interface Cat extends Mammal, Animal

There is more on interfaces later on in this lesson.

### An abstract classes

An abstract class represents a slightly more granular concept of another concept but still not specific enough for a concrete representations. So in out **Cat** hierarchy we would design abstract classes to represent the Genus. The Genus has some characteristics and behaviours that are common for that grouping of cats. Therefore the **Panthera** genus is represented as an abstract class that implements the **Cat** interface.

public abstract class Panthera implements Cat {}

It implements that **Cat** interface, by using the **implements** keyword, to show that this is the type of family it belongs to and to provide a very general way to refer to the **Panthera** and the concrete **Tiger** and **Leopard** class. You will be seeing much more about why this is important later on in this section. For now just think of the **Panthera** class as a type of **Cat**.

The **Panthera** is still a concept of a **Tiger** or **Leopard**, although more specialised than the **Cat** interface it still is not an actual cat, so it does not make sense to be able to create a concern instance of this class. If I were to attempt to create an instance of the Panthera abstract class with the new keyword the code would not compile.

**Panthera panthera = new Panthera (); // will not compile**

The **Panthera** genus has characteristics that are common among all **Tigers** and **Leopards** so it make send to code those commonalities in the abstract class.

In this code snippet below I have code some of the common characteristics that all **Pantheras** have.

public abstract class Panthera implements Cat {  
  
 public String lifeStyle = "Nocturnal";

public void roar(){  
 System.*out*.println("Roars");  
 }  
  
 public abstract void hunt();  
  
  
}

As you can see I have code that the lifestyle of all **Pantheras** is nocturnal. This has been codes as a public instance **String**. Also I have code a public method that represents that all **Pantheras** roar. I am simplifying my model a little from reality and assuming that all **Panthera** make exactly the same roaring sound. This has been implemented as a public method that just prints **“roar”** to the console. And finally I have code a method that represents that **Pantheras** hunt. Now in this case I want to represent that **Pantheras** can hunt but the way in which each type of **Panthera** (i.e. **Tiger**, **Leopard**) hunt in different way. Perhaps **Tigers** hunt in packs and **Leopards** hunt on their own (again I am taking liberties with real-life). I create an abstract method to code a situation where a behaviour exists but the exact nature of that behaviour is dependent on the actually species itself (subclass). I use the abstract keyword to create the abstract method.

public abstract void hunt();

Abstract methods don’t have a method body; this is in order to indicate that the behaviour of the method is specific to the concrete class that extends it (more on this in a moment). Abstract methods are always public, protected or package private (i.e. no keyword is used).

There is more on abstract method later on in this lesson.

### The concrete class

The concrete class is the representation of an actual **Cat** which might be a **Tiger**, **Leopard** etc. The concrete class must extend the abstract class from which it inherits all the characteristics of the Genus. The code below shows the concrete implementation of the **Tiger** class.

public class Tiger extends Panthera {  
  
 protected void hunt() {  
 System.*out*.println("Hunt in packs ");  
 }  
  
}

The **Tiger** class extends the abstract **Pantera** class with the keyword extends. The **Tiger** class inherits the methods and properties of the abstract class and the obligation to provide a concrete implementation of the **hunt()** method. As you can see the **hunt()** methods container behaviour hat is specific to the **Tiger**. The **Leopard** concrete path also has the same obligation to provide an implementation for the **hunt()** method.

public class Leopard extends Panthera {

protected void hunt() {  
 System.*out*.println("Hunt on its own");  
 }

}

As you can see its implementation is different because the hunt behaviour of the **Leopard** is not the same as the hunt behaviour of **Tiger**.

### Subclassing

The concept of subclassing is about creating a new class and extending the capabilities of another class. As you have seen so far a class can subclass another class with the keyword **extends** followed by the class name. This way to subclass can only be done if the class to extend is another class. It can be either a concrete class or an abstract class and only one class can be extended. The following code would not compile because it tries to extend two classes, **Pathera** and **Carnivorous**:

public class Tiger extends Panthera, Carnivorous

A class cannot extend from two or more other classes. This is called multiple inheritances and is not permitted in the java language. Java is a single inheritance language.

The subclass is also often referred to as the child class of the class it extends and the class from which it extends is often referred to as the parent class.

Child classes only have access to the public, protected and package private methods of the parent class. In the following example the **Panthera** class has a private field speed which is not accessible from the child class **Tiger**.

public abstract class Panthera implements Cat {  
  
 private int speed = 45;  
  
 //…  
  
}

public class Tiger extends Panthera {  
  
 public static void main(String... args) {  
 System.*out*.println("The tiger's speed is: " + speed);  
 }  
  
}

The **Tiger** class will fail to compile. To access the speed variable the parent class must provide a method that returns the variables value.

### The Object class

The **java.lang.Object** class is the implicit parent of all classes, (not interfaces). It is the ultimate parent and even though you do not explicitly extend this class it is extend automatically by the compiler. If your class extends another class then the parent class will be made to extend **Object**. If that class extends another class then that parent will extend **Object** and so on up the class hierarchy until the top where **Object** will always be implied as the grandparent of all classes.

In the example of the **Cat** class hierarchy the **Pantera** abstract class extends the **Object**.

public abstract class Panthera extends Object implements Cat {

### Inherited methods and fields

In the hierarchy we have created the **Cat** interface is at the top of the hierarchy tree. The **Panthera** abstract class implements the **Cat** interfaces and the **Tiger** concrete class extends the **Panthera** abstract class and inherits its methods and fields and has to provide a concrete implementation of the **hunt()** method. Even though you cannot see the **lifestyle** field and the **roar()** method in the **Tiger** class itself they are there and behaviour as if you had copied then into the class manually. Let’s see an example of how this works.

public class Tiger extends Panthera {  
   
 protected void hunt() {  
 System.*out*.println("Hunt in packs ");  
 }  
  
 public static void main(String... args) {  
 Tiger tiger = new Tiger();  
 String lifestyle = tiger.lifeStyle;  
 System.*out*.println("The tigers lifestyle is: " + lifestyle);  
 tiger.hunt();  
 tiger.roar();  
 }  
  
}

I have added a main method to the **Tiger** class that accesses the **lifestyle** field and executes the two methods: **hunt()** and **roar()**. When you run the code snippet the output shows that the **roar()** method and **lifestyle** fields are part of the **Tiger** class.

The tigers lifestyle is: Nocturnal

Hunt in packs

Roars

### Class access modifiers

Classes can be defined with access modifiers just like method can and they can be: **public**, **protected**, package private, **private** and **final**. However you only need to know about the **public**, **final** and package private modifiers for the exam. The **protected** and **private** modifiers are used with inner classes, and beyond the scope of the exam.

A Java file can have only one public class but it can contain main package private classes as shown in this example.

public class OneFileManyClasses{}  
class ClassOne{}  
class ClassTwo{}  
class ClassThree{}  
interface AnInterface{}  
abstract class AnAbstractClass{}

As you can see there is only one public class but many package private classes (concrete, abstract and interface). The name for the file will be **OneFileManyClasses.java**.

## Order of keywords

The order in which the class keywords are placed in important. The example below shows clearly the order in which they must appear. If the order is not observed the class with fail to compile

[access modifier] + **abstract** + **class** + [className] + **extends** + [className] + **interface** + [interfaceNames]

Here is an example of what it might look like.

public abstract class Panthera implements Cat

For an interface the order is as follows:

[access modifier] + **interface** + [interfaceName] + **extends** + [interfaceNames]

Here is an example of what it might look like.

public interface Cat extends Mammal, Animal

Remember that an interface can extend multiple interfaces but it cannot extend classes and it cannot implement other interfaces.

## Overriding and overloading

A child class inherits all public and protected methods (and fields) from the parent class. With this feature comes the option to override or overload those inheritance classes. What this means is that you can redefine the behaviour of those inheritance classes and provide functionality that is more relevant to your class. Let’s start by looking at overriding a method.

## Method selection

### Most specific parameter matching

When the compiler sees methods with parameters that are compatible such as an **int** and an **Integer** the compiler choses the method with most specific type. Take a look at this example.

public void eat(int calories){}  
public void eat(Integer calories){}

The two method are overloaded so when calling the eat() method which one is chosen. The compiler will not autobox a call if the parameter is a primitive. The method chosen when called by **eat(100)** is the method that accepts the integer primitive as this is the most specific type the matches to argument type.

If there were no method with the primitive integer type the compiler would autobox the integer, promoting it to an **Integer** type, and it would select the **eat(Integer calories)** method.

### Method with reference parameters

The same rules of selecting the most specific type is also true for method parameters that are reference types. Take a look are this example.

public void eat(Leopard leopard){}  
public void eat(Panthera panthera){}  
public void eat(Cat cat){}

A call to the eat method will select the method which has the type the most specifically matches that of the parameter type. The following code calls the method with the Leopard parameter.

Leopard leopard = new Leopard();  
eat(leopard);

If there were no method with a Leopard parameter the method with the Panthera parameter would be selected instead and if there were no methods with the Panthera or Leopard parameter then the method with the Cat parameter would be selected.

### Methods with primitives parameters

Selection of methods with primitive parameters following the same rules as for methods with reference parameters. An exact match will be made if possible, if not, a larger type will be selected.

public void eat(long calories){}

A call to the **eat(10)** method will select **eat(long calories)**. This is because an integer can be implicitly cast to a long. The reverse will not happen as a cast down can only be done explicitly and the compiler will not do that. So the call **eat(10)** will not call the method **eat(short calories),** even though the number 10 can be cast to a short without loss of precision.

## Hiding methods and variables

### Hiding static methods

A hidden static method is a method that has an equivalent method in the child class with the same name, parameter list and a covariant return type. The access modifier must be at least as restrictive and it must not throw an exception that is new or wider.

Take a look at this example.

public abstract class Puma implements Cat {  
  
 public static String type(String message){  
 return message.toLowerCase();  
 }  
  
}

public class Cougar extends Puma {  
  
 public static String type(String message){  
 return message.toUpperCase();  
 }  
  
}

The static **type** method in the **Cougar** child class hides the static **type** method in the **Puma** parent class.

## Contrast hidden methods and overridden methods

The essential difference is the selection of the method when a call is made to that method. The question that is answered is: is the parent or child method chosen when it is referenced. The answer is not as simple as saying that it’s the method in the class from where the method call originates.

Take a look at an example of method hiding.

public abstract class Puma implements Cat {  
  
 public static String type() {  
 return "Puma";  
 }  
  
 public void printParentType() {  
 System.*out*.println("Parent type is: " + *type*());  
 }  
  
}

public class Cougar extends Puma {  
  
 public static String type(){  
 return "Cougar";  
 }  
  
 public void printChildType() {  
 System.*out*.println("Child type is: " + *type*());  
 }  
  
 public static void main(String... args){  
 Cougar cougar = new Cougar();  
 cougar.printChildType();  
 cougar.printParentType();  
 }  
  
}

The output is:

**Child type is: Cougar  
Parent type is: Puma**

This shows that the **type()** method called is determined by the class in which the call originates. This is quite logical. However the story is different when the methods are overridden.

Take a look at an example of method hiding.

public abstract class Puma implements Cat {  
  
 public String type() {  
 return "Puma";  
 }

}

public class Cougar extends Puma {

public String type(){  
 return "Cougar";  
 }

}

The type() method is now overridden in the Cougar class. If I run the example again the output is quite different.

**Child type is: Cougar  
Parent type is: Cougar**

As you can see the **type()** method chosen is determined by the type of the reference used to make the first method call. This is a feature of a concept known as polymorphism. We will cover this topic later in the lesson.

### Hiding variables

You can define a variable in a child class that has the same name and type. This is called variable hiding. Variables can never be overridden and the rules for selecting are that the class from which the call to the variable originates determined the variable selected.

public abstract class Panthera implements Cat {  
  
 public String lifeStyle = "Nocturnal";  
  
  
}

public class Leopard extends Panthera {  
  
 public String lifeStyle = "Dusk";  
  
  
}

A method called from within the **Panthera** class that uses the **lifeStyle** variable will have the value **“Nocturnal”** and likewise, a method called from within the **Leopard** class uses the **lifeStyle** variable that has the value **“Dusk”**.

# Subsection 2: Learn abstract classes and interfaces

## Abstract classes

The idea of an abstract class is to allow a partial implementation to be built, from which other developers can add or override default behaviour.

We have seen use of abstract class already in this lesson and in the first part we talked about it with regard the **Cat** analogy. In this section I am going to cover a few more details and reiterate some of the rules for abstract classes.

### Interface fields

Interface fields are assumed to be public, static and final so can only carry the public access modifier. If the modifier is not present it is deemed to be present. The fields can be marked static and final. They cannot be private, protected or package private or abstract and because they are static they behave in the same way as any other static variable. We discussed this earlier on in the course.

### Default Methods

A default method in an interface is a method with a default implementation. A class can choose to override the method but is not obliged to do so. If it is not overridden then the method is inherited.

A default method can only be declared within an interface. If you attempt to define one in an abstract class or a concrete class compilation will fail.

There are rules for a correctly constructed default method:

* It must carry the default keyword
* It must have a method body, even though it can be an empty body
* It is assumed to be public so is not required to have the public keyword
* It cannot be private, protected, static, final or abstract
* There can be multiple or no default method in an interface
* If overridden the rules for overriding methods must be adhered to

Watch out for clashes between default methods in multiple interfaces that have the same signature. If there is a clash the compilation will fail. However if the default method is overridden then the overridden class is used as there is no clash as the overridden method is always invoked.

### Static interface method

Interfaces can have static methods. The behaviour is similar to static methods in classes, except that it is not inherited by any class that implements the interface. So the only way to call static method in the interface is to use the interface name directly. As shown in this example.

interface Cat {  
  
 static void walk() {  
 System.*out*.println("Car is breathing!");  
 }

}

Cat.*walk*();

The static method cannot be private or protected as it is assumed to be public.

# Subsection 3: Access objects and constructors using super and this

You have already learnt in a previous lesson that classes have constructors that are used to build the class instance. When dealing with classes that extend from a parent the constructors become a little bit more involved.

### Constructor requirements and behaviours

* The first line of a constructor is an implicit call to **super()** if no explicit call to **super()** or **this()** is present. A call to **super()** or **this()** cannot be on the second or later lines. Compilation will fail.
* The compiler will insert a no argument constructor on the first line if no explicit call to **super()** is made. If the parent class does not have a default constructor compilation will fail.
* The **super()** call invokes a constructor in the parent class. It can invoke a constructor that has arguments by passing the **super()** call the appropriate arguments. E.g. **super(5**) calls the parent constructor which has a integer as the argument.

### Accessing inherited fields and methods

A child class may access the public, protected and default package (if child and parent are in the same class) fields and methods of the parent class. Access to private fields and methods is not allowed as we saw earlier in the lesson. The value of private fields is only accessible via a method that exposes the value such as a public getter method that returns the private fields value.

To access the value of a field in a parent (or any class in the hierarchy of classes) you simple use the variable as shown in this code snippet.

public class Leopard extends Panthera {  
  
 public void printLifeStyle() {  
 System.*out*.println("Leopard is " + lifeStyle);  
 }  
  
 public static void main(String... arg) {  
 new Leopard().printLifeStyle();  
 }

}

You can also use the **this** keyword (don’t confuse it with **this()**) to refer to the current or parent instance of the class. So in the above example the code in the **printLifeStyle()** method could be expressed as follows:

public void printLifeStyle() {  
 System.*out*.println("Leopard is " + this.lifeStyle);  
}

The value of lifestyle is taken from the parent class, although technically it is actually a member of the child class too.

Strictly speaking the **this** keyword in not necessary in most cases and is only used when there are ambiguities with conflicting variable names. You saw an example of this in a previous lesson when talking about constructors.

There is another keyword called **super** that represents the parent class instance only. Let’s see an example.

Image that I add a **lifeStyle** variable to the **Leopard** class as shown in the example below.

public class Leopard extends Panthera {  
  
 public String lifeStyle = "Dusk";  
  
 public void printLifeStyle() {  
 System.*out*.println("Leopard is " +lifeStyle);  
 }  
  
}

What will be the value of **lifeStyle** printed to the console by the **printLifeStyle()** method. Will it be **Dusk** or will it be **Nocturnal**?

The answer is that is it will be **Dusk**. The implicit **this** keyword refers to the current class’s instance. If we want to access the parent class’s **lifeStyle** instance we can use the keyword **super** to refer to the parent class’s instance. The **printLifeStyle()** method now looks like this and **Nocturnal** is printed to the console.

public void printLifeStyle() {  
 System.*out*.println("Leopard is " + super.lifeStyle);  
}

Remember that this can refer to the parent and child class’s instance but **super** can only refer to the parent class’s instance. Also don’t get confused between **super()** and **super** and **this** and **this()** they are not interchangeable. The following code snippet will not compile.

public void printLifeStyle() {  
 System.*out*.println("Leopard is " + super().lifeStyle);  
}

# Subsection 4: Write polymorphism code

The concept of polymorphism is one at the heart of the Java programming language. It is where one object can behave like any of the classes in its hierarchy.

# Subsection 5: Use casting correctly

When you pass an object reference to a method that accepts a super type, access to the original type’s methods are lost and restricted to the methods defined in the super type (and its super types).

# Subsection 6: Common APIs and lambdas expression

During this course you have met many core APIs such as ArrayList and String and you have used some of the functionality of these classes. You are expected to know about and how to use the course classes and data structures commonly used in the Java API.

In this section I will introduce you to the core APIs that you should know. I will not go through every method with a demonstration because I think that the best way to learn how each method and class work is to write you own code and try it out for yourself.

The APIs that you should know are:

* java.lang.String
  + https://docs.oracle.com/javase/8/docs/api/java/lang/String.html
* java.lang.StringBuilder
  + https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html
* java.util.Arrays
  + https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html
* java.util.ArrayList
  + https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
* all the wrapper classes in java.lang package such as java.lang.Integer
  + <https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html> ( all the other wrapper classes that extend from Number)
  + <https://docs.oracle.com/javase/8/docs/api/java/lang/Number.html>
* java.time.format.DateTimeFormatter
* java.time.\* (including Period, LocalDate, LocalTime and LocalDateTime)
  + <https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html>
  + <https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html>
  + <https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html>

## Lambdas and Predicates

A lambda expression is a block of code that gets passed around. They are snippets of functionality that perform a function that can be generalised.

## Lambda syntax

Lambdas have various syntax that you should be familiar with. Here are examples of valid lambda expression:

(Cat cat) -> System.*out*.println("Bandage");  
(Cat cat, Dog dog) -> System.*out*.println(cat.type() + " loves " + dog.type());  
(Cat cat) -> System.*out*.println("Bandage");  
(Cat cat) -> {};  
() -> {};  
() -> false;