Chapter 13
Classes, Inheritance and Abstraction

13.1 Introduction

Inheritance is one of the most powerful features of Object Orientation. It is the difference between an encapsulated language that provides an object-based model and an Object-Oriented language. Inheritance is also one of the main tools supporting reuse in an Object-Oriented language (although in Scala's case Traits are also a major tool for reuse). You will use inheritance all the time without even realising it, indeed you have already been doing so every time you have benefited from the default implementation of `toString`.

Scala has single class inheritance although it does have a form of multiple inheritance via Traits.

13.1.1 What Are Classes for?

In some Object-Oriented languages, classes are merely templates used to construct objects (or instances). In these languages, the class definition specifies the structure of the object and a separate mechanism is often used to create the object using this template.

In some other languages (e.g. Java, C#, Smalltalk), classes are objects in their own right; this means that they can not only create objects, they can also hold data, receive messages and execute methods just like any other object. However, many programmers find this distinction confusing and Scala has adopted the Companion object approach instead (see later in this book) which means that a class is supported by a singleton object that can hold data and provide a placeholder for additional supporting behaviours.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_13

Thus, in Scala, classes are unique within a program and can:

- defined using the keyword class,
- be used to create instances (via the keyword new),
- be inherited by subclasses (and can inherit from existing classes),
- mix in traits,
- define properties,
- define methods,
- define functions,
- define instance variables and values,
- be sent messages.

Objects, on the other hand, are:

- defined using the keyword object,
- singleton entities within the systems,
- cannot be instantiated (and do not support the *new* operation),
- cannot be used to create new instances,
- are accessed directly via their name rather than via any val or var.

Confusingly many Object-Oriented languages use the term object to refer to an instance of a class. In Scala an instance of a class is exactly that, an instance of a class, an object *is a different concept.* Instances can be

- created from a class (using the keyword new),
- hold their own copy of their state (in terms of properties or instance variables),
- be sent messages,
- execute instance methods,
- execute functions,
- have many copies in the system (all with their own data).

13.2 Inheritance Between Types

To recap on the concept of inheritance, inheritance is supported between types within Scala. For example, a class and extend (subclass) another class. A trait (another type) can extend other traits, etc., and objects can extend traits or classes. All of these types support and enable inheritance.

In terms of the inheritance we say:

- A subtype inherits from a super type
- A subtype obtains all code and data from the super type
- A Subtype can add new code and data
- A subtype can override inherited code and data
- A subtype can invoke inherited behaviour or access inherited data.

13.3 Inheritance Between Classes

Inheritance is achieved in Scala using the `extends` keyword (as was discussed in Chap. 6). Scala is a single class inheritance system, so a Scala class can only inherit from a single class (although it can mix in multiple traits, to be discussed later).

The following class definition builds on the class `Person` presented earlier:

```
class Person(val name: String, var age: Int)

class Student(var subject: String,
              n: String,
              a: Int)
  extends Person(n, a) {

}
```

This class extends the class `Person` by adding a new variable property, `subject`. As this is a **var** the class `Student` also provides reader and writer functionality for the `subject` property. We say that `Student` is a subclass of `Person` and that `Person` is the super class of `Student`.

Note that as the class `Person` defined two properties in its primary constructor, the class `Student` must invoke the constructor explicitly. It does this by indicating the data to pass to this constructor after the parent class name following the extends expression. For the student class we take these values in as part of its own constructor. However the parameters 'n' and 'a' are not properties; they are local fields which can be used within the definition of the class Student. We are only using them to pass the data up to the definition of the constructor in the class Person. As a result you can only instantiate the class Student by providing the *subject* to be studied, the student's *name* and their *age*. For example,

```
object StudentTest extends App {
  val s = new Student("Computer Science", "John", 18)
  println(s.name + " is studying " + s.subject)
}
```

The end result is that a new instance of the class `Student` is created that has a `subject` property and also a `name` property and an age property *inherited* from the class `Person`. In fact the instance referred to by the variables is a `Student` and is also a Person (in the same way that any human is also a Mammal, etc.).

Note that it is necessary to invoke a parent class's constructor explicitly. The only exceptions to this are if the parent class only defines a Zero parameter constructor, or if the primary constructor provides default values for all of its parameters.

13.3.1 The Role of a Subclass

There are only a small number of things that a subclass should do relative to its parent or super class. If a proposed subclass does not do any of these, then your selected parent class is not the most appropriate super class to use.

A subclass should modify the behaviour of its parent class or extend the data held by its parent class. This modification should refine the class in one or more of these ways:

- Changes to the external protocol, the set of messages to which instances of the class respond.
- Changes in the implementation of the methods, i.e. the way in which the messages are handled.
- Additional behaviour that references inherited behaviour.

If a subclass does not provide one or more of the above, then it is incorrectly placed. For example, if a subclass implements a set of new methods, but does not refer to the instance variables or methods of the parent class, then the class is not really a subclass of the parent (it does not extend it).

As an example, consider the class hierarchy illustrated in Fig. 13.1. A generic root class has been defined. This class defines a `Conveyance` which has doors, fuel (both with default values) and a method, `startUp`, that starts the engine of the conveyance. Three subclasses of `Conveyance` have also been defined: `Dinghy`, `Car` and `Tank`. Two of these subclasses are appropriate, but one should probably not inherit from `Conveyance`. We shall consider each in turn to determine their suitability.

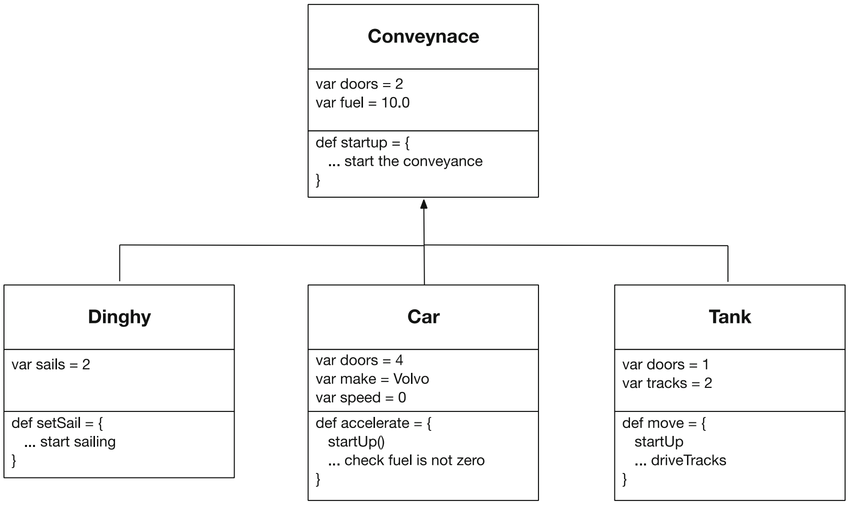


Fig. 13.1 A class and its subclasses

The class `Tank` overrides the number of doors inherited, uses the `startUp` method within the method `fire` and provides a new instance variable. It therefore matches all three of our criteria.

Similarly, the class `Car` overrides the number of doors and uses the method `startUp`. It also uses the instance variable `fuel` within a new method `accelerate`. It also, therefore, matches our criteria.

The class `Dinghy` defines a new instance variable `sails` and a new method `setSail`. As such, it does not use any of the features inherited from `Conveyance`. However, we might say that it has extended `Conveyance` by providing this instance variable and method. We must then consider the features provided by `Conveyance`. We can ask ourselves whether they make sense within the context of `Dinghy`. If we assume that a dinghy is a small sail-powered boat, with no cabin and no engine, then nothing inherited from `Conveyance` is useful. In this case, it is likely that `Conveyance` is misnamed, as it defines some sort of a motor vehicle, and the `Dinghy` class should not have extended it.

The exceptions to this rule are subclasses of `Any` and `AnyRef`. This is because these classes are the root types in the Scala type hierarchy. `AnyRef` is the root of all reference types—that is classes in Scala. As you must create a new class by subclassing it from an existing class, you can subclass from `AnyRef` when there is no other appropriate class.

13.3.2 Capabilities of Classes

A subclass or class should accomplish one specific purpose; it should capture only one idea. If more than one idea is encapsulated in a class, you may reduce the chances for reuse, as well as contravene the laws of encapsulation in Object-Oriented systems. For example, you may have merged two concepts together so that one can directly access the data of another. This is rarely desirable.

Breaking a class down costs little but may produce major gains in reusability and flexibility. If you find that when you try and separate one class into two or more classes, some of the code needs to be duplicated for each class, then the use of abstract classes can be very helpful. That is, you can place the common code into an abstract superclass to avoid unnecessary duplication.

The following guidelines may help you to decide whether to split the class with which you are working. Look at the comment describing the class (if there is no class comment, this is a bad sign in itself). Consider the following points:

- Is the comment short and clear. If not, is this a reflection on the class? Consider how the comment can be broken down into a series of short clear comments. Base the new classes around those comments.

- If the comment is short and clear, do the class and instance variables make sense within the context of the comment? If they do not, then the class needs to be re-evaluated. It may be that the comment is inappropriate, or the class and instance variables inappropriate.

Look at the instance variable references (i.e. look at where the instance variable access methods are used). Is their use in line with the class comment? If not, then you should take appropriate action.

13.3.3 Overriding Behaviour

As was mentioned at the start of this chapter, a subtype (e.g. a subclass) can override the behaviour defined in a parent class. In fact it is possible to override both methods and fields. It should be noted that in Scala; it is also possible to override a parameterless method by a new field or property (this is actually to do with the way in which Scala internally represents data and methods) but can be useful and also confusing.

To override either a field or a method in a parent class you must use the keyword override. You have seen this already with the `toString` method where we had to include the keyword override in order to redefine `toString` to do something more useful then display the fully qualified class name and a hexadecimal number. Of course the default behaviour of `toString` was being inherited into our classes via the class `AnyRef` (which we implicitly extended).

In the following example, the class `Base` overrides `toString` so that the name and age properties of the `Base` class are used to create the string representation of instances of the class. It also defines a method max and a property working.

```
class Base(val name: String, var age: Int) {
  def max(x: Int, y: Int): Int = if (x > y) x else y
  val working = false
  override def toString() = s"$name is $age"
}
```

We can then subclass `Base` with the class `Derived` and override both `max` and `working` if we wish, for example,

```
class Derived(n: String, a: Int) extends Base(n, a) {
  override def max(x: Int, y: Int): Int =
    if (x > y) y else x
  override val working = true
}
```

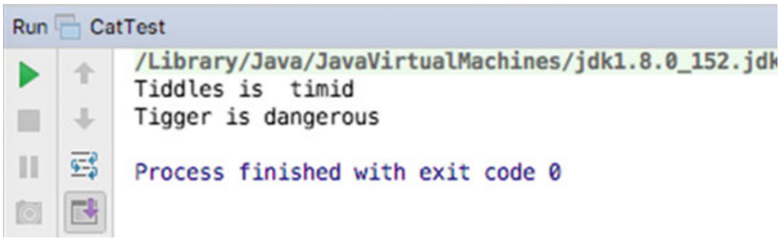
In `Derived` we have redefined max to actually return the minimum value for some reason and overridden working to be true.

As another option consider the classes `Cat` and `Tiger` below

- Cat has *vals* `dangerous` and `name`.
- Tiger overrides `dangerous` and `name`. However, the value for `name` is now set when the instance is created. Thus the property that is defined as part of the constructor overrides a property used with the `Cat` class, which was not originally part of any construction process.

```
class Cat {
  val dangerous = false
  val name: String = "Tiddles"
  override def toString =
    s"$name is ${(if (dangerous) "dangerous" else " timid")}"
}

class Tiger(override val name: String) extends Cat {
  override val dangerous = true
}

object CatTest extends App {
  var c = new Cat()
  println(c)
  c = new Tiger("Tigger")
  println(c)
 }
```

The effect of running the `CatTest` program is shown below.



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk
Tiddles is  timid
Tigger is dangerous

Process finished with exit code 0
```

13.3.4 Protected Members

By default, within Scala all behaviour (methods and functions) as well as data (properties) is public, that is they are visible (can be accessed) anywhere within an application. We have seen that it is possible to mark both behaviour and data as private so that they are only accessible within a single object or class. However, there is another option which has not been mentioned yet. That is, it is possible to make either behaviour or data *protected*.

Protected members of a class are members (methods, functions, properties) that can only be accessed in the current class and in subclasses *and* only in subclasses. They are not visible to other elements of an application.

For example, in the following abstract class `Base` the property age is public, the method `max` is public and the overridden method `toString` is public. However, the property working is only visible within `Base` and any subclasses of `Base`.

```
class Base(val name: String, var age: Int) {
  def max(x: Int, y: Int): Int = if (x > y) x else y
  val working = false
  override def toString() = s"$name is $age"
}
```

The use of protected properties or behaviour helps to explicitly specify the interface between a subtype and its super type.

13.4 Restricting a Subclass

You can restrict the ability of a subclass to change what it inherits from its superclass. Indeed, you can also stop subclasses being created from a class. This is done using the keyword `final`. This keyword has different meanings depending on where it is used. For example in the following example, the keyword final has been applied to the whole class:

```
final class Employee(n: String,
                     a: Int,
                     company: String)
  extends Person(n, a)
```

This means that no element of this class can be extended, so no subclass of `Employee` can be created.

The keyword final can also be applied to a public property. For example,

```
final var maximumMemory = 256
```

This indicates that the property `maximumMemory` cannot be overridden in a subclass. This means that the value of `maximumMemory` is set for this class and for all subclasses wherever they are defined by this class. Using a `val` instead of a `var` means that the value cannot merely be overridden by a subclass, it is also only set once and is thus a constant for the hierarchy below the current class:

```
class Employee(n: String,
               a: Int,
               company: String)
  extends Person(n, a) {

  final val max = 10
}
```

The keyword `final` can also be applied to methods. This means that a method cannot be overridden in a subclass, for example,

```
class Volunteer (n: String ,
                a: Int,
                company: String)
  extends Person(n, a)  {

  final def prettyPrint(): Unit = {
    println("Volunteer")
    println("\tName: " + name)
    println("\tAge: " + age)
    println("\tCompany: " + company)
      }

    }
```

This states that the method `prettyPrint` cannot be overridden in a subclass. That is, a subclass cannot redefine `prettyPrint();` it must use the one that it inherits.

Restricting the ability to overwrite part of, or all of, a class is a very useful feature. It is particularly important where the correct behaviour of the class and its subclasses relies on the correct functioning of particular methods, or the appropriate value of a variable, etc. A class is normally only specified as `final` when it does not make sense to create a subclass of it. These situations need to be analysed carefully to ensure that no unexpected scenarios are likely to occur.

13.5 Abstract Classes

An *abstract* class is a class from which you cannot create an object. It is missing one or more elements required to create a fully functioning instance. In contrast a non-abstract (or concrete) class leaves nothing undefined and can be used to create a working instance. You may wonder what use an abstract class is. The answer is that you can group together elements that are to be shared amongst a number of classes, without providing a complete implementation. In addition, you can force subclasses to provide specific methods ensuring that implementers of a subclass at least supply appropriately named methods. You should therefore use abstract classes when:

- you wish to specify data or behaviour common to a set of classes, but insufficient for a single instance,
- you wish to force subclasses to provide specific behaviour.

In many cases, the two situations go together. Typically, the aspects of the class to be defined as abstract are specific to each class, while what has been implemented is common to all classes.

For example, consider the following class.

```
abstract class Person(val name: String, var age: Int) {
  // Override inherited toString
override def toString = s"$name, $age";

  //Define an abstract method
  def prettyPrint
  def birthday = age = age + 1
}
```

This is a revised version of the Person class we have seen several times before. However we are now making Person an abstract concept. This means that you do not create instances o the Person class itself, but rather you create instances of subclasses of Person such as Employee, Student, Graduate, etc. Person brings together the common features of these subclasses, but on its own it is not sufficient to warrant an instance being created. It is only the concrete classes (non-abstract classes) which actually make sense as instances:

This abstract class definition means that you cannot create an instance of `Person`. Within the definition of `Person,` we can see that the `toString` and birthday methods are concrete or defined methods, whereas the method `prettyPrint` is not defined (it has no method body). The `prettyPrint` method is what is known as an abstract method. Any class, which has one or more abstract methods, is necessarily abstract (and must therefore have the keywords `abstract class`). However, a class can be abstracted without specifying any abstract methods.

An abstract class can also define any number of concrete methods. The method birthday is a concrete method that adds one to the current age of the person.

Any subclass of `Person` must implement the `prettyPrint` method if instances are to be created from it. Each subclass can define how to *pretty print itself* in a different manner. The following `Graduate` class provides a concrete class that builds on `Person`:

```
class Graduate(n: String, a: Int,
              degree: String,
              uni: String) extends Person(n, a) {

  val institution: String = uni

  def this(n: String, a: Int, degree: String) =
                          this(n, a, degree, "Oxford")

  override def toString =
    s"Graduate ${super.toString} $degree ]";

  def prettyPrint = {
    println("Graduate")
    println("\tName: " + name)
    println("\tAge: " + age)
    println("\tDegree: " + degree)
    println("\tUniversity: " + uni)
  }
  }
```

This class extends the class `Person` and also provides:

- The four-parameter constructor is used to passing the name and age for the `Person` class's primary constructor and to provide a degree and University for the `Graduate` class.
- A three-parameter auxiliary constructor invokes the four-parameter primary constructor.
- A concrete version of the `prettyPrint` method.

We can also return to the `Employee` class from earlier and see that it also provides a concrete `prettyPrint` method and invokes the `Person` class's primary constructor:

```
class Employee(n: String, a: Int, company: String)
                              extends Person(n, a) {

  final def prettyPrint(): Unit = {
    println("Employee")
    println("\tName: " + name)
    println("\tAge: " + age)
    println("\tCompany: " + company)
  }

}
```

13.6 The Super Keyword

We have already seen that it is possible to override behaviour defined in a parent class so that the version in the current class meets that needs of that class. The method `toString` is a typical example of this. In numerous examples we have redefined `toString` to create a string based on the data held by a class rather than to use the generic version. To do this we used the keyword override and ensured that the method signature (its name, parameters and return matched those defined in the parent class).

However, rather than completely override the version of the method defined in the parent class we can chose to extend its behaviour. This is done by defining a new version of a method in a subclass and then using the keyword *super* to invoke the version defined higher up the inheritance hierarchy.

For example, in the following example, the abstract class Base defines a method print that prints out a message "Base print". The subclass Derived extends Base and overrides the method print. However within the body of the method it called `super.print` that causes it to invoke the parent class's version of print. Note this call could be made anywhere within the body of the method print in Derived, it does *not* need to be the first line of the method.

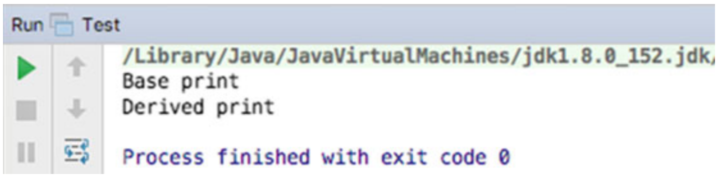
```
abstract class Base {
  def print = println("Base print")
}

class Derived extends Base {
  override def print {
    super.print
    println("Derived print")
  }
}
```

The effect of the overridden print method in `Derived` is that it calls the parent class's version of print. This means that in effect it extends, rather than replaces, the behaviour of the original version of print. Note that super tells Scala to start searching up the class hierarchy for a version of print defined above the current class in the hierarchy. In this case it is defined in the parent class, but it could have been defined in a parent of `Base`—that is it starts searching Base and will continue search up the class hierarchy until it finds another definition of print to execute.

To illustrate this idea we could create a simple application:

```
object Test extends App {
  var d = new Derived()
  d.print
}
```

If we now run the above application, the output would be as shown below:



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
Base print
Derived print

Process finished with exit code 0
```

Here you can see that both the original version and the derived version of print have been executed.

13.7 Scala Type Hierarchy

The type hierarchy in Scala is complicated by the presence of traits, but the core types are divided between two types, AnyVal and AnyRef, with the class Any at the root (see Fig. 13.2).

Thus the root of everything in Scala is the abstract class Any. Any has two subclasses, the abstract AnyVal and the concrete AnyRef:

- **AnyVal** this is used to represent Value like types, such as Boolean, Char, Byte, Short, Int, Long, Float, Double. Strictly speaking Scala has no primitive types—these are objects. However, they are a special type of objects that are managed by the Scala runtime efficiently.
- **AnyRef** this is used for all reference types such as classes and traits. Examples of AnyRef subtypes include the data structure (or collection) classes such as Array, List, Seq and String. It is also used as the root for all user-defined classes that do not explicitly extend any other class.

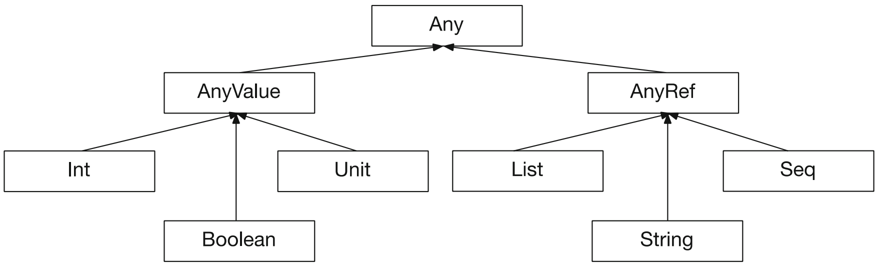


Fig. 13.2 Simplified extract form Scala type hierarchy

13.8 Polymorphism

Polymorphism was a concept discussed earlier in the book relating to one of the four key concepts in Object Orientation. In terms of Scala programming Polymorphism means that a *val* or *var* local variable or property can actually reference an instance of a particular type or any subtype of that type. Thus a var of type Person can actually hold a reference to a Person (assuming it is not an abstract type) or any subclass of Person (including Student, Employee and graduate, etc.).

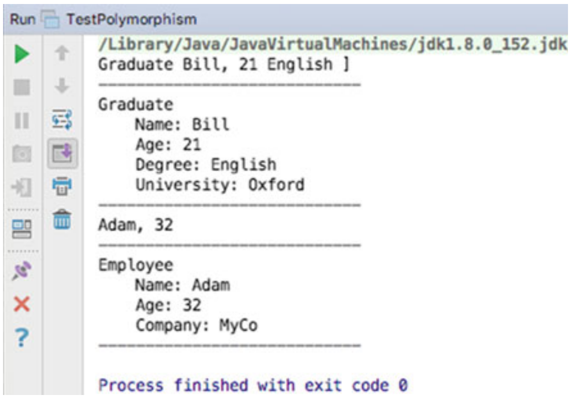
For example, we can write:

```
object TestPolymorphism extends App {
  var p: Person = new Graduate("Bill", 21, "English")
  println(p)
  println("---------------------------")
  p.prettyPrint
  println("---------------------------")
  p = new Employee("Adam", 32, "MyCo")
  println(p)
  println("---------------------------")
  p.prettyPrint
  println("---------------------------")
}
```

In this test application the variable p is of type `Person`. It can thus reference a `Person`, a `Graduate` or an `Employee`. Initially we are storing a reference to a `Graduate` in p. We then call `println` on p (which causes the `toString` method to be invoked on the instance reference by p and then call `prettyPrint` on `p`. Both `toString` and `prettyPrint` can be guaranteed to be available in whatever instance `p` refers to because the functionality in the class Person guarantees it. Any methods defined only in the `Graduate` are not visible via `p` (although they are still present in the instance being referenced, they just cannot be accessed at this point).

After this we create a new instance of the `Employee` class and store a reference to that instance in p and then use `toString` to print the object out and `prettyPrint` again. However, the behaviour that now executes is whatever behaviour is either defined in `Employee` or inherited into `Employee`.

The output produced by this application is shown here.

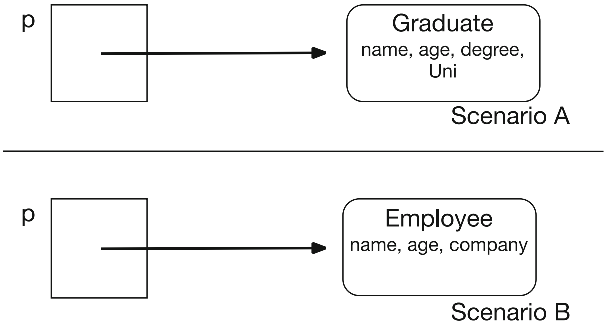


```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk
Graduate Bill, 21 English ]
------------------------------
Graduate
    Name: Bill
    Age: 21
    Degree: English
    University: Oxford
------------------------------
Adam, 32
------------------------------
Employee
    Name: Adam
    Age: 32
    Company: MyCo
------------------------------

Process finished with exit code 0
```

The key here is that with polymorphism:

1. The type of the variable p acts as a filter—ensuring that only common behaviour is accessible
2. But at runtime the actual definition of, for example, `prettyPrint` is dynamically bound. That is, the version defined the class that the instance is actually an example of what is executed.

The illustrate sees the following diagram, and scenario A indicates the situation when p references a graduate. Thus when the `prettyPrint` method is called on p at that point, it is the Graduate version of `prettyPrint` that is run. Scenario B indicates the situation when p references an employee. Thus when `prettyPrint` is called on this instance, it is the version in Employee that is `run`.



To summarise then, polymorphism in Scala is similar to that in languages such as Java and C#, in that:

- A variable of type X can refer to instance of X or any subclass of X
- At runtime method invocations are dynamically bound based on the type of the receiving object (not the type of the variable)