Outline

- Review of Definitions in Scala
  - Classes, Traits, Objects
    - Constructors and constructor parameters
  - Def, (lazy) val and var
- Cases classes and pattern matching
  - Examples
  - Library Examples: Option, List
- Inheritance, extensibility and mixins
- Visibility modifiers in Scala
  - Private, protected, qualified private and protected
Classes, Traits and Objects

- All have same kinds of members
  -Defs, vars and vals.
  -Type synonyms.
  -Nested class definitions

- Classes define a template for objects
  -Instantiated via a new class expression

- Classes and traits may have type parameters
  -Object definitions may not

- Traits (mixins) define members just like classes
  -Used to define interfaces and as reusable components for defining classes via inheritance

- Object definitions name a singleton object
  -Classes may have companion objects
    -Same name with special access rights
    -c.f. use of statics in Java
Constructors

- Classes may have constructor arguments.
  - Traits and objects may not.

- Constructor arguments define fields
  - Only visible to this instance of the class, by default
  - May be declared explicitly as val or var
    - So can be made public (or private or protected …)

- Each class has a primary constructor
  - A class may have auxiliary constructors, of form
    \[ \text{this} (\text{ps}_1) \ldots (\text{ps}_n) = e \]
    - Each auxiliary constructor must invoke a preceding constructor as the first statement in its definition \( e \)

- Constructors are not inherited
Def declaration for methods

- Return type declaration is optional
  - Except for abstract or recursive methods
- Argument types must be declared
- Definition bodies have different forms
  - \( = e \)
    - Type is determined by type of \( e \)
  - \( = \{ \ldots \} \)
    - Type is determined by last expression of block
  - \( \{ \ldots \} \)
    - Is of type \texttt{Unit} (c.f. \texttt{void} in Java)
- Defs without bodies are abstract
  - i.e. method declarations without definitions
Val declarations

- Val declares a constant value associated with a class instance (i.e. object)
  - This provides per object constants
    - like final instance fields in Java
- Vals may be
  - **Abstract**, with no initialising expression
    - val limit
  - **Defined**, with an initialising expression
    - val limit = 1024
  - **Lazy**, with an initialising expression
    - lazy val limit = ackerman (5)
    - Which is not evaluated (and stored) until its first use
**Var declarations**

- Var declares a variable associated with a class instance (i.e. object)
  - like instance fields in Java
  - Actually Scala vars are more like Java Bean properties
    - They have associated getter and setter methods

- Declaring `var x: T` is equivalent to declaring
  - a getter function `x`, and
  - a setter function `x_=`,
  - declared as follows:
    - `def x: T`
    - `def x_= (y: T): Unit`
Var definitions

- Variable declarations may have accompanying definitions
- Vars may be explicitly initialised
  - `var x: T = e`
  - The type `T` is optional
- Or their getters and setters may be explicitly defined
  - `def x: T = e`
  - `def x_ = (y: T): Unit = … some update …`
- This allows definitions to check validity of assignments
- BUT user code looks like a simple assignment
Example:
from Scala Reference manual

class TimeOfDayVar {
    private var h: Int = 0
    private var m: Int = 0
    private var s: Int = 0

    def hours = h
    def hours_=(h: Int) =
        if (0 <= h && h < 24)
            this.h = h
        else
            throw new DateError()

    def minutes = m
    def minutes_=(m: Int) =
        if (0 <= m && m < 60)
            this.m = m
        else
            throw new DateError()

    def seconds = s
    def seconds_=(s: Int) =
        if (0 <= s && s < 60)
            this.s = s
        else
            throw new DateError()

}  // end of TimeOfDayVar

val d = new TimeOfDayVar
    d.hours = 25 // throws a DateError exception
    d.hours = 8
    d.minutes = 30
    d.seconds = 0

}
Example:
from Scala Reference manual

class TimeOfDayVar {
  private var h: Int = 0
  private var m: Int = 0
  private var s: Int = 0

  def hours = h
  def hours_=(h: Int) =
    if (0 <= h && h < 24)
      this.h = h
    else throw new DateError()

  def minutes = m
  def minutes_=(m: Int) =
    if (0 <= m && m < 60)
      this.m = m
    else throw new DateError()

  def seconds = s
  def seconds_=(s: Int) =
    if (0 <= s && s < 60)
      this.s = s
    else throw new DateError()

  val d = new TimeOfDayVar
  d.hours = 8
  d.minutes = 30
  d.seconds = 0
  d.hours = 25 // throws a DateError exception
}
Overriding in subclasses

- Definitions may be overridden in subclasses
- Types must be consistent
  - See later for more details on how types may be varied in subclasses
- Keyword `override` must be used
  - Tells compiler to check consistency of override
  - Otherwise may end up with a distinct overloaded method
    - i.e. a method with same name but different signature
- A no-arg `def` may be overridden by a `val` (!)
  - Allows subclasses to store instead of compute result of a superclass `def`
Uniform Access Principle

- Principle identified by Bertrand Meyer of Eiffel programming language (and design by contract) fame

- Same syntax is used for
  - No-arg method calls
  - Data (var or val) access

- In Scala when we use code like
  - `target.access`
  - we cannot tell whether the access invokes a no-arg method, or is a direct access to data

- Thus client code is not sensitive to this design decision
  - So changing the decision is OK
  - An overridden def may be a val
Outline

- Review of Definitions in Scala
  - Classes, Traits, Objects
    - Constructors and constructor parameters
  - Def, (lazy) val and var
- Cases classes and pattern matching
  - Examples
  - Library Examples: Option, List
- Inheritance, extensibility and mixins
- Visibility modifiers in Scala
  - Private, protected, qualified private and protected
Case Classes

- In Scala case classes allow us to split a superclass abstraction into special classes
  - This is a special form of subclassing
    - Usually we define functions over the abstraction
  - In other languages this facility is like datatype definitions
    - using disjoint sums (ML or Haskell)
    - or unions (C, rarely used)
Discussion on Inheritance

- Inheritance of classes and traits allow us to extend and specialise classes.
- Normally a subclass should define objects with at least the same capability, and behaviour, as a superclass.
  - The *Liskov substitutability principle* says that clients should be able to use subclass objects as if they were superclass objects.
- Introductory OO texts describe subclassing as providing an *is-a relationship* between types of objects.
  - QUESTION: is a square a rectangle?
Design by Contract (DbC)

- Class invariants specify validity constraints on the state of objects of a class
- Preconditions specify applicability conditions for method calls
  - Allowable state for target object and
- Postconditions specify effects of methods
  - Constraints on return values and updates on object states
- OO Design hint:
  - Even if you do not adopt DbC formally in your OO designs, you should use it conceptually to help maintain the integrity of your (re-)designs
Subcontracting with DbC

- Subclasses inherit contracts:
  - Invariants may be strengthened
  - Preconditions may be weakened
  - Postconditions may be strengthened

- Rationale:
  - Liskov substitutability principle

- This allows client code to reason about behaviour using the contract of a superclass
  - When the actual implementation is provided by a superclass
Uses of Inheritance

- In practice we use inheritance in different ways
  - Implementation of higher level interfaces and/or abstract classes
    - Alternative subclasses may provide different implementations with different data structures
    - Subclasses may provide their own implementation invariants, but a correct implementation will make sure that client code is not affected by built-in constraints
  - Extension / refinement of existing classes
    - Adding extra data and methods allows us to refine the state space, and to override existing method implementations
    - Every instance of a superclass may have more than one corresponding instance of a subclass
      - Subclass invariants capture more information than superclasses.
Uses of Inheritance …

- Specialisation / restriction of existing classes
  - In a subclass we may restrict allowable object states
    - Impose a stronger class invariant in a subclass
  - This allows us to model a superclass type of object as a “sum” or choice of subclass types
    - i.e. we can split an object state into different cases
- Note
  - This may imply that inherited methods may not always be applicable
    - i.e. inherited preconditions of methods may be strengthened
  - This breaks the subcontract rule of design by contract
    - And the Liskov substitutability principle

- In programming with case classes we usually have to handle all cases at once
  - This limits extensibility and reusability of our code, but is often more convenient
A Comment on OO Design

- Objects are defined in classes
  - ... defining the type of object
- Objects do no change their types over their lifetime
- Classes define what kind of operations an object can undergo
  - e.g. a Square class defines a particular type of 2D shape, with stronger constraints than a Rectangle
  - A Rectangle can sensibly support more operations than a Square e.g. *stretch* one side (but not *skew*)
- Class hierarchies for mutable objects tend to be flatter than for immutable ones
Case Class Examples

- Expression **Tree** split into cases
  - **Var**, **Sum**, **Count**
  - **eval** method for expressions is defined outside of the data type definitions
    - More functional/procedural style than OO style

- Demo + code walk through
  - See previous slides (1.oofp) for (similar) code
    - and Java version in OO style, with eval method belonging to the Tree class
    - Simple Practice Exercise: rewrite Java version in Scala

- **Complex2** demo
  - Complex numbers as **Rect** or **Polar** implementations
Scala Library Examples

- **Option[A]** split into **None** and **Some[A]**
  - Used extensively to capture optional values `a` of type `A`.
    - Use this in preference to (often ambiguous) null returns
    - To construct a value, simply write the pattern `Some (a)`
  - **None** is an object
    - Both objects and case classes can be pattern matched
    - Actually any class instance can be if an appropriate “deconstructor” is provided via an `unapply` method def

- Browse code or API doc to see utility methods

- Aside: for Haskell programmers, think Maybe!
  - As in Haskell, Option implements monadic operations to allow special kinds of iterations
    - `for` syntax in Scala, `do` syntax in Haskell
Scala Library Examples

- **List[A]** split into non-empty list cases **Nil** and non-empty list **head :: tail**.
  - Construct lists by adding head element to a tail list
  - Pattern matching definitions over lists
  - Note the use of symbolic name (::) as a class name!
  - Also recall that symbolic names ending in : yield right associative operators
    - \( 1 :: 2 :: 3 :: 4 :: \text{Nil} =\ 1 :: (2 :: (3 :: (4 :: \text{Nil}))) \)
      \( =\ \text{Nil} :: (4) :: (3) :: (2) :: (1) \)
      \( =\ \text{List (1, 2, 3, 4)} \)
- Scala programming idiom: Companion objects often provide factory methods for their classes
  - **List (1, 2, 3, 4)** is defined because the companion object **List** has an appropriate **apply (a: A*)** method defined
Outline

- Review of Definitions in Scala
  - Classes, Traits, Objects
    - Constructors and constructor parameters
  - Def, (lazy) val and var
- Cases classes and pattern matching
  - Examples
  - Library Examples: Option, List
- Inheritance, extensibility and mixins
- Visibility modifiers in Scala
  - Private, protected, qualified private and protected
Inheritance in Java

- Inheritance in Java limits code reuse
  - Classes extend most one superclass
    - Single inheritance of classes
  - Interfaces may extend many interfaces
  - Classes may implement many interfaces
    - Multiple inheritance of interfaces
- BUT interfaces may only contain method declarations
  - Method code can only be defined in classes
- So in Java there is only one inheritance slot available for inheriting code
Mixins with Scala

- Scala also uses single inheritance of classes and “multiple inheritance” of traits
- BUT Scala allows the same forms of definition in both classes and traits
  - This allows code to be inherited via multiple sources
- Traits are used
  - To declare interfaces, as in Java
  - To define small reusable code components
    - Also called mixins
  - Mixin composition combines multiple traits
    - E.g. class C is a subclass of D and combines traits $T_1 \ldots T_n$

class C extends D with $T_1$ with $T_2$ with ... with $T_n$
Rules of Mixin Composition

- What if the same method is overridden in multiple traits?
  - Mixins are stacked
  - Mixins to the right take effect last
    - There is a precise linearisation defined for the ancestors of a class
  - This influences the way in which super calls may be nested in a stack of overrides for a method
    - The linearisation determines what super refers to
Linearisation of Mixins

- Example from Programming in Scala chpt 12

```scala
class Animal
trait Furry extends Animal
trait FourLegged extends Animal with HasLegs
trait HasLegs extends Animal
class Cat extends Animal with Furry with FourLegged
```

<table>
<thead>
<tr>
<th>Class</th>
<th>Linearization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animal</td>
<td>Animal, AnyRef, Any</td>
</tr>
<tr>
<td>Furry</td>
<td>Furry, Animal, AnyRef, Any</td>
</tr>
<tr>
<td>FourLegged</td>
<td>FourLegged, HasLegs, Animal, AnyRef, Any</td>
</tr>
<tr>
<td>HasLegs</td>
<td>HasLegs, Animal, AnyRef, Any</td>
</tr>
<tr>
<td>Cat</td>
<td>Cat, FourLegged, HasLegs, Furry, Animal, AnyRef, Any</td>
</tr>
</tbody>
</table>

Table 12.1: Linearization of several classes

- The order in which traits are listed may change the meaning of the mixin
Outline

- Review of Definitions in Scala
  - Classes, Traits, Objects
    - Constructors and constructor parameters
  - Def, (lazy) val and var
- Cases classes and pattern matching
  - Examples
  - Library Examples: Option, List
- Inheritance, extensibility and mixins
- Visibility modifiers in Scala
  - Private, protected, qualified private and protected
Visibility of Class Members

- As in Java we can control name visibility using access modifiers such as protected and private
  - The rules are different in Scala, with extra control via qualifiers
- Default access is public with no access limitation. There is no public keyword.
- **Private** means access is limited to current class
  - And the companion object
  - Unlike Java, access to private members does not extend to nested class members
Visibility ...

- **Protected** means access is limited to current class and any descendant classes
  - Unlike Java does not extend to same package
- **Qualified access** modifiers give more control over access by naming a visibility scope
  - May be a *package*, a *class*, *this*
    - `private [pkg_name]` means that access is granted to any member of the package
    - `private [class_name]` means that access is granted to any member of the class, including nested classes
    - `private [this]` restricts access to the current instance of the class (i.e. it’s an object not class level control)
    - Similarly qualified protected extends access to classes or packages