Class Hierarchies

Abstract Classes

- Consider the task of writing a class for sets of integers with the following operations:

```
abstract class IntSet:
  def incl (x: Int): IntSet // returns the existing set with x added
  def contains (x: Int): Boolean // checks if x is in the existing set
```

- IntSet is an *abstract class*
- Abstract classes can contain members which are missing an implementation; these are called *abstract members*
- Consequently, no direct instances of an abstract class can be created (for instance, an IntSet() call would be illegal)

Class Extensions

- Let's consider implementing sets as binary trees
- There are two types of possible trees: a tree for the empty set, and a tree consisting of an integer and two sub-trees
- Here are their implementations:

```
class Empty() extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty(), Empty())

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet:
  def contains(x: Int): Boolean =
    if (x < elem) left.contains(x)
    else if (x > elem) right.contains(x)
    else true
  def incl(x: Int): IntSet =
    if (x < elem) NonEmpty(elem, left.incl(x), right)
    else if (x > elem) NonEmpty(elem, left, right.incl(x))
    else this
```

  - ***Empty*** and ***NonEmpty*** both ***extend*** the class ***IntSet***
  - This implies that the types ***Empty*** and ***NonEmpty*** conform to the type IntSet and implement all its abstract methods

Base Classes and Subclasses

- *IntSet* is called the **superclass** of *Empty* and *NonEmpty*
- *Empty* and *NonEmpty* are **subclasses** of *IntSet*
- In Scala any user-defined class extends another class
- If no superclass is given, the standard class *Object* in the Java package java.lang is assumed
- The direct or indirect superclass of a class C are called **base classes** of C

Implementation and Overriding

- The definitions of *contains* and *incl* in the classes *Empty* and *NonEmpty* **implement** the abstract functions in the base trait *IntSet*

- It is also possible to redefine an existing, non-abstract definition in a subclass by using ***override***
- Writing the redefinition of an existing, non-abstract method without override would give an error; The reason why you are forced to do that is that the compiler wants to make sure you don't have an accidental collision where you just define a method, think it's a new method, but that method accidentally replaces a method in the subclass, so override is essentially an opt in marker that says, that's what I intend

Object Definitions

- When you create a class that only needs one instance to be created, you can define it as a **singleton object** using the *object definition:*

```
object Empty extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
```

- Singleton objects are values, so the class evaluates itself

Companion Objects

- An object and a class can have the same name; this is possible since Scala has two global **namespaces**: one for types and one for values
- Classes live in the type namespace, while Objects live in the values namespace
- If a class and an object with the same name are given in the same source file, we call them **companions**
- A companion object of a class plays a role similar to static class definitions in Java (which are absent in Scala)

Programs

- It is possible to create standalone applications in Scala
- Each such application contains an object with a main method
- Once this program is compiled, you can start it from the command line with:

```
scala prog_name
```

- Writing main methods is similar to what Java does for programs
- Scala has also a more convenient way to do it: a standalone application is alternatively a function that's annotated with ***@main*** and that can take command line arguments as parameters
- Once this function is compiled, you can start it from the command line

Dynamic Binding

- Object-oriented languages implement **dynamic method dispatch**
- This means that the code invoked by a method call depends on the runtime type of the object that contains the method (the sequence of reduction that gets performed depends on the value on the left hand-side of the method)