1. Subtyping and Contravariance

(a) \( f \) could call its function argument on any \( \text{Shape} \), e.g. either \( \text{Circle} \) or \( \text{Rectangle} \). Thus, calling \( f \) on a function of type \( \text{Rectangle} \rightarrow \text{Int} \) is not allowed, because \( \text{Rectangle} \rightarrow \text{Int} \) is not a subtype of \( \text{Shape} \rightarrow \text{Int} \). If this call was executed, then \( f \) could call its argument on a \( \text{Circle} \), which would not match the expected \( \text{Rectangle} \) argument type.

(b) \( g \) can only call its function argument on a \( \text{Circle} \). Thus, calling \( g \) on a function of type \( \text{Shape} \rightarrow \text{Int} \) is allowed, because \( \text{Shape} \rightarrow \text{Int} \) is a subtype of \( \text{Circle} \rightarrow \text{Int} \). If we execute this call, then whatever \( g \) does with its function argument will be fine, since the expected type of the function argument is \( \text{Shape} \), so it can handle any particular type of shape such as \( \text{Circle} \).

2. Modules and Interfaces in Scala

(a) The components are accessed as follows:

\[
\begin{array}{lllllll}
\text{A.c} & \text{A.d} & \text{A.f} & \text{B.c} & \text{B.d} & \text{B.f}
\end{array}
\]

(b) After the two import statements, \( d \) refers to the string value \( \text{B.d} = "1234" \) since this was the most recent import. If we import in the opposite order it refers to \( \text{A.d} = 2 \).

(c) The trait should be something like:

```
trait ABlike {
  type T
  val c: T
  val d: T
  def f(x: T, y: T): T
}
```

(d) \( \text{def g(x: ABlike) = x.f(x.c,x.d)} \)

According to the Scala interpreter the return type is \( x.T \).

(e) \( \text{g(new ABlike{}} \)

```
  type T = Boolean
  val c = true
  val d = false
  def f(x: T, y: T) = x && y
})
```

3. Type parameters

(a) \( \text{abstract class Tree[A]} \)

```
\text{case class Leaf[A](a: A) extends Tree[A]}
\text{case class Node[A](t1: Tree[A], t2: Tree[A]) extends Tree[A]}
```
def sum(t: Tree[Int]) : Int = t match {
  case Leaf(a) => a
  case Node(t1,t2) => sum(t1) + sum(t2)
}

def map[A,B](t: Tree[A])(f: A => B): Tree[B] = t match {
  case Leaf(a) => Leaf(f(a))
  case Node(t1,t2) => Node(map(t1)(f), map(t2)(f))
}

def flatten[A](t: Tree[Tree[A]]): Tree[A] = t match {
  case Leaf(u) => u
  case Node(t1,t2) => Node(flatten(t1),flatten(t2))
}

def flatMap(t: Tree[A])(f: A => Tree[B]) = flatten(map(t)(f))

4. (∗) Ad hoc polymorphism

(a)

abstract class List[A] extends HasSize

  case class Nil[A]() extends List[A] {
    def size() = 0
  }

  case class Cons[A](head: A, tail: List[A]) extends List[A] {
    def size() = tail.size() + 1
  }

(b)

abstract class Tree[A] extends HasSize

  case class Leaf[A](a: A) extends Tree[A] {
    def size() = 1
  }

  case class Node[A](t1: Tree[A], t2: Tree[A]) extends Tree[A] {
    def size() = t1.size() + t2.size()
  }

(c)

def sameSize(x: HasSize, y: HasSize) = x.size() == y.size()

scala> sameSize(Cons(1,Nil()), Leaf("abc"))
res2: Boolean = true