1. Subtyping and Contravariance

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

(b) $g$ can only call its function argument on a Circle. Thus, calling $g$ on a function of type $\text{Shape} \to \text{Int}$ is allowed, because $\text{Shape} \to \text{Int}$ is a subtype of $\text{Circle} \to \text{Int}$. 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 Shape, so it can handle any particular type of shape such as Circle.

2. Modules and Interfaces in Scala

(a) The components are accessed as follows:

```scala
A.c A.d A.f B.c B.d B.f
```

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

(c) The trait should be something like:

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

(d) `def g(x: ABlike) = x.f(x.c,x.d)`

According to the Scala interpreter the return type is $x.T$.

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

3. Type parameters

(a) `abstract class Tree[A]
case class Leaf[A](a: A) extends Tree[A]
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 {
  Leaf(u) => u
  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