Exercise 9.2

- Objects with many different properties can be represented much easier in an OODB than in an RDB. In an RDB, one would need either many different tables for the different object types or have a few tables where the tuples may have many NULL values.

- A class hierarchy is a more natural way to represent different types of toys and parts than multiple relations (i.e. specialisation hierarchies can be represented).

- In an RDB, objects will be distributed across multiple tables. Thus, updating and maintaining the data is more difficult in a relational system, because one has to update multiple tuples at the same time.

- Relationships between objects can be represented by pointers in an OODB. In an RDB one has to use foreign keys and joins which is not good for the performance of the system.

- In OODBs, complex object types (e.g. sets, lists, nested objects) can be used to represent complex “real world” objects.

- Some basic functionality for the management of the data of the toys can be encapsulated in the definition of the object types, so it is in the database. In a relational system, such functionality has to be included in the application. Thus, development of applications might be easier.

- Schema evolution: changes to the schema of RDBs are difficult, they sometimes requires the changes in the application programs. This may be easier in an OODB because we just need to add new classes, attributes, or operations.
Exercise 9.4

a)

type Person is
public name;
body [ name : string ; ]
operations ...
implementation ...
end type Person;

type Employee supertype Person is
public getBoss;
body [ boss : Manager ; ]
operations
     declare getBoss: → Manager;
implementation
     define getBoss is
         return self.boss;
end type Employee;

type Manager supertype Employee is
body [ boss2 : CEO ; ]   !! Note: Attributes cannot be refined
operations
     refine getBoss: → CEO;
implementation
     define getBoss is
         return self.boss2;
end type Employee;

type CEO supertype Manager ...
end type CEO;

b) This is how the operations should be defined:

declare hire: Manager || Employee → void;
refine hire: CEO || Manager → void;

As Manager.hire requires an argument of type Employee, it can also be of type Manager and CEO. So the declaration of the function does not imply a restriction on their application. Furthermore, the argument type of the refined operation (Manager) is subtype of the original argument type (Employee). This is also not allowed in strong-typed languages as GOM.
Reason: otherwise loss of substitutability: Consider a CEO-object that is assigned to variable of type Manager. The compiler can only guarantee that objects of class Employee are passed as arguments to hire method calls. This must be avoided if the object assigned to the variable is in fact a CEO.
c) This is how the operations should be defined:

\begin{verbatim}
declare setBoss: Employee || Manager → void;
refine setBoss: Manager || CEO → void;
\end{verbatim}

This is the same problem as before. If we refine an operation, then the argument types have to be the same type or supertype of the original argument types (contravariance). Note, that other OO-languages such as Java and C++ do not allow to refine operations. These languages allow only overloading of operations.
Exercise 9.5

a)

Otherwise loss of substitutability.
Consider two classes:

```
type A is
  public x
  body [ x: T; ]
  operations...
  implementation ...
end type A;
```

```
type B supertype A is
  public x
  body [ x: Tsub; ]
  operations...
  implementation ...
end type B;
```

Assume, \( T_{sub} \) is a subtype of \( T \).
Consider the following code snippet:
```
var  anA: A;
anB: B;
anT: T;
begin
  anA <- anB;
  anA.x <- anT;  // ok for compiler, but shouldn't be allowed since anA.x is in fact of type \( T_{sub} \).
end;
```

Assume, we refine the \( x \) attribute in \( B \) as being of type \( T_{super} \) (supertype of \( T \)).
Then the following code snippet shows the loss of substitutability:
```
var  anA: A;
anB: B;
anT: T;
begin
  anA <- anB;
  anT <- anA.x  // ok for compiler, but shouldn't be allowed since anA.x is in fact of type \( T_{super} \).
end;
```

This shows, that retyping attributes in strong typed languages is not possible.
You can also easily understand this, if you remember, that for a public attribute GOM implicitly declares VCO and VRO operations. The has a variable of the attribute’s
type a an parameter whereas the VRO returns an object of the attributes' class.
Now remember the rules for signature refinement (slides 62): As the type of the retyped attribute occurs as an argument and as a result type in the derived class at
the same time, it must be sub- and super-typ of the original class.

b) with the explanations before: Yes, read-only attributes can be retyped to sub-types
(compare rules for VRO).

c) Explicitly declare the VRO (e.g. as getBoss in 9.3a)
Exercise 9.6

a) cuboid is subtype of cube

```plaintext
type cube is  
  public length;  
  body [ length : float; ] 
operations  
  declare volume: → float; 
  declare rotate: float → void; 
implementation ...
end type cube;
```

```plaintext
type cuboid supertype cube is  
  public width, height; 
  body [ width : float; 
         height : float; ] 
operations  
  declare volume: → float; 
  declare rotate: float → void; 
implementation ...
end type cuboid;
```

Advantage:
- each object type has exactly the attributes and methods which are required

Problems:
- not much reusability of functions: most functions for cuboid have to be redefined

- **semantic**: cuboid is a subtype of cube means that every cuboid is a cube (which is obviously not the case). Therefore, we can pass a cuboid as an argument to a function where a cube is expected, e.g. Pipe.connect(Cube). This will lead to unexpected/undefined results.

b) cube is subtype of cuboid

```plaintext
type cuboid is  
  public length, width, height;  
  body [ length : float; 
         width : float; 
         height : float; ] 
operations  
  declare volume: → float; 
  declare setWidth: float → void; 
implementation ...
end type cuboid;
```

```plaintext
type cube supertype cuboid is  
end type cube;
```

Advantages:
- specialisation is represented according to the extension of the classes in the real world (i.e. all cubes are also cuboids).

- operations for cuboid can be reused by cube. However, it might be more efficient to overwrite some operations because they are simpler for a cube (e.g. rotate).
Problems:
- Cube inherits attributes (width + height) which are not necessary
  - Program must maintain consistency (length=width=height)
  - Requires more memory.
- Cube inherits operations which are not (or should not be) applicable to a cube, e.g. setWidth.

Summary
- There is no common solution to such problems:
  It depends on the context and the requirements which solution is better. As database designer, you might want to choose option a) as it requires less memory. As a programmer (e.g. in Java or C++), you might choose option b) as it ensures that a method gets the correct object type as argument. Consistency of the attributes is maintained by the implementation of the cube class.