Summary: This topic focuses on types, and the subtyping relationship between types, along with the related concept of when one kind of object can be substituted for another.
Multiple inheritance emulation in full

The following is a complete example of emulating multiple inheritance using delegation, following from the previous notes.

- The new method implementation for savings is in a new class SavingsDelegate that doesn't inherit from Account.
- Instead, the SavingsAccount class contains a SavingsDelegate, to which it delegates the addInterest method.
- The delegate constructor is passed the SavingsAccount object so that it updates the balance for the right account.
- We add rawDeposit to avoid tax on interest deposits.

```java
public class Account {
    private int balance;
    public int getBalance() { return getBalance(); }  // The getBalance() method is declared but not implemented.
    public void rawDeposit(int amount) { balance += amount; }
    public void deposit(int amount) { rawDeposit(amount); }
    public void withdraw(int amount) { balance -= amount; }
}

public class SavingsDelegate {
    private int interestRate=6;
    private Account acct;
    public SavingsDelegate(Account acct) { this.acct = acct; }
    public void addInterest() {
        acct.rawDeposit(acct.getBalance() * interestRate/100);  // The getBalance() method is not implemented.
    }
}

public class SavingsAccount extends Account {
    private SavingsDelegate delegate = new SavingsDelegate(this);
    public void addInterest() {
        delegate.addInterest();  // Delegation
    }
}
Multiple Inheritance Emulation: CheckSavings

- The ChequeAccount class is exactly as before.
- ChequeSavings inherits from ChequeAccount, and reuses the implementation for addInterest by delegating this method to a SavingsDelegate object, exactly as in SavingsAccount.
- Thus, the implementation of this method only needs to be in one place, and is reused appropriately instead of being repeated.
- The “glue” code for delegation is repeated, but it will always be quite short, and is unlikely to be changed.

```java
public class ChequeAccount extends Account {
    private static final int depositTax = 1;
    public void deposit(int amount) {
        super.deposit( amount*(100-depositTax/100) );
    }
    public void processCheque(int amount) { withdraw(amount); }
}

public class ChequeSavings extends ChequeAccount {
    private SavingsDelegate delegate = new SavingsDelegate(this);
    public void addInterest() { delegate.addInterest(); }
}
```
Types

Java is a statically typed language.

- This means that the types of variables and methods are checked for consistency at compile time (or even in an IDE).

- If a variable or a method is used in a way that is inconsistent with its type, then the programmer is alerted so that they can fix the issue.

- This is a much more efficient than resolving such issues by running the program and having it crash or misbehave due to an inconsistency.

- Dynamically typed languages are still popular for some purposes such as scripting where programs are not too large.

Many early object oriented languages were dynamically typed – including Smalltalk, the first general purpose OO language.

- This was partly due to objects requiring more a complicated form of types compared to previous non-OO languages.

- The design of types for objects is still an active research area.

- Type systems for generics with objects were not in the original definition of Java, and the theory behind them has been developed since.
Why bother with fancy types for objects?

The Java language could be greatly simplified by having only a single type for all objects, but still with different classes as now.

- All classes would then have instances with this type “Object”.
- All current Java programs would still compile and run.
  - The types for objects would always be consistent because there's only the single type “Object”.
- But, writing programs would be more difficult.
  - If you passed the wrong type of object to a method, you wouldn't find out until you ran the program, and at some point an attempt is made to call a method on an object that lacks it.
- For this reason, Java and most other popular OO languages include more complex types for objects.
- Object-oriented languages did not become mainstream until such complex types were introduced in languages like C++ and Java.
Subtyping

The basic concept of object types is that there is one type for each class and interface.

The key complication with object types is the need for subtyping between types.

The motivation for subtyping comes from the following argument.

- Instances of the subclass must possess all data areas associated with the parent class.
- Instances of the subclass must implement, through inheritance at least (if not explicitly overridden) all functionality defined for the parent class. (They can also define new functionality, but that is unimportant for the present argument).
- Thus, an instance of a child class can mimic the behavior of the parent class. It therefore seems reasonable that a variable declared as a parent, should be able to hold a value generated from the child class.

The *principle of substitutability* is sometimes called *Liskov substitutability*, since one of the first people to describe the idea was Barbara Liskov, of MIT.
Subclass vs Subtype

Of course, the problem with this argument is that a child class can override a method and make arbitrary changes. It is therefore useful to define two separate concepts:

- To say that A is a subclass of B merely asserts that A is formed using inheritance.
- To say that a is a subtype of B asserts that A preserves the meaning of all the operations in B.

It is possible to form subclasses that are not subtypes; and (in some languages at least) form subtypes that are not subclasses.

How can something be a subclass but not a subtype?

What is a type really anyway?

What do we mean when we use the term type in describing a programming language?

- A set of values. (The type int, for example, describes -2147483648 to 2147483647)
- A set of operations. (We can do arithmetic on ints, not on booleans).
- A set of properties. (If we divide 8 by 5 we are not surprised when the result is 1, and not 1.6).

What about when we consider classes (or interfaces) as a system for defining types?
The Problem of Defining Types

Consider how we might define a Stack ADT:

```java
interface Stack {
    public void push (Object value);
    public Object top ();
    public void pop ();
}
```

Notice how the interface itself says nothing about the LIFO property, which is the key defining feature of a stack. Is the following a stack?

```java
class NonStack implements Stack {
    public void push (Object value) { v = value; }
    public Object top () { return v; }
    public void pop () { v = null; }

    private Object v = null;
}
```

The Definition of Subtype

So now we can better understand the concept of a subtype.
A subtype preserves the meaning (purpose, or intent) of the parent.
The problem is that meaning is extremely difficult to define. Think about how to define the LIFO characteristics of the stack.
Subclasses are not Necessarily Subtypes

It is easy to create a subclass that is not a subtype -- think of \texttt{NonStack}. It is also possible to create subtypes that are not subclasses – in Java you can do this using interfaces.

- In other languages it is even possible to have classes that result in subtypes of other classes, without inheritance.

The Substitution Paradox

There is a curious paradox that lies at the heart of most strongly typed object-oriented programming languages.

- Substitution is permitted, based on subclasses. That is, a variable declared as the parent type is allowed to hold a value derived from a child type.
- Yet from a semantic point of view, substitution only makes sense if the expression value is a subtype of the target variable.

If substitution only makes sense for subtypes and not for all subclasses, why do programming languages based the validity of assignment on subclasses?
The Undecidability of the Subtype Relationship

It is trivial to determine if one class is a subclass of another. It is extremely difficult to define meaning (think of the Stack ADT), and even if you can it is almost always impossible to determine if one class preserves the meaning of another. One of the classic corollaries of the halting problem is that there is no procedure that can determine, in general, if two programs have equivalent behavior.

Is This a Problem?

What does it take to create a subclass that is not a subtype?

- The new class must override at least one method from the parent
- It must preserve the type signatures
- But it must violate some important property of the parent

Is this common? Not likely. But it shows you where to look for problem areas.
**Summary for Subtyping vs Subclassing**

- To say a class is a subclass of another simply asserts that is was built using inheritance.
- By itself, the subclass relationship says nothing about the behavior of the child in relation to the parent.
- The term subtype is used to describe a class that matches the behavior of another class.
- It is easy to build subclasses that are not subtypes. It is possible (although not as easy) to build subtypes that are not subclasses.