Remember Abstract Data Types (ADTs)

- Combining data with associated operations

Haskell:

```haskell
module Stack (Stack, empty, isEmpty, push, top, pop) where

empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a, Stack a)
```

Remember Record-Subtyping

\[
\begin{align*}
\{x: \{a: \text{Int}, b: \text{Bool}\}, y: \{m: \text{Int}\}\} & \prec \{x: \{a: \text{Int}\}, y: \{\}\} \\
\{i_1 \in \text{Int}\} & \subseteq \{k_j \in \text{Int}\} \\
& \vdash k_j = i_1 \rightarrow \star S_i < : T_j \\
& \vdash \{k_1 \in S_j \in \text{Int}\} \prec \{k_1 \in T_j \in \text{Int}\} \\
& \text{for each } i : \Gamma \vdash \star t_i : T_i \\
& \Gamma \vdash \star \{i_1 = t_1, \ldots, i_n = t_n\} : \{i_1 : T_1, \ldots, i_n : T_n\} \\
\end{align*}
\]

Objects

- Class = Template from which Objects can be created
- Object = Instance of a Class.

- Encapsulation
  - Data is hidden from direct manipulation from the outside

- Subtyping

- Inheritance
  - Subclasses inherit behavior from the superclass, can selectively override where desired

- Dynamic Dispatch
  - Object determines which implementation of a polymorphic method is called
Encapsulation

- Underlying principle: Modularity
  - Minimize interdependencies among Modules
  - Maximizing reuse

- Data abstraction
  - External interface describes abstract methods

- The “Closed” in the Open-Closed Principle
  - Class should be open for extension but closed for modification

Example

- Java:
  ```java
  public class Stack <E> {
    private ArrayList<E> list = new ArrayList<E>();
    public void push(final E elem) {
      list.add(0, elem);
    }
    public E pop () {
      return list.remove(0);
    }
  }
  ```

Subtyping

- We know subtyping for functions:
  - Contravariance for arguments of subtype
  - Covariance for return type of subtype

- These rules are syntactical

Behavioral subtyping

- Behavioral subtyping is semantic:
  - The two syntactical rules (for methods)
  - No new exceptions can be thrown by the subtype
  - Preconditions cannot be strengthened by the subtype
  - Postconditions cannot be weakened by the subtype
  - Invariants must be preserved
  - History constraint: New methods introduced by the subtype may not allow state changes that were not allowed in the methods of the supertype
Liskov Substitution Principle

- If we have strong behavioral subtyping, the following (called Liskov Substitution Principle or LSP) holds:
  
  - Let $q(x)$ be a property provable about objects $x$ of type $T$. Then $q(y)$ should be provable for objects $y$ of type $S$ where $S$ is a subtype of $T$.

Inheritance

- Reusing functionality by specializing a superclass

- Creates a hierarchy of classes which are in an “is-a” relationship

- Derived classes can override methods of the superclasses.

Example:

```java
class A {
    int a;
    float b;
    void foo();
}

class B extends A {
    double c;
}
```
**Subtyping != Inheritance**
- Subtyping requires structural equivalence
  - Only then we can safely substitute
- Inheritance creates a nominal subtype which may or may not also be a structural subtype
- Many language restrict inheritance to what is permissible as structural subtypes, hence the confusion.
- Eiffel and C++ are examples of languages which allows inheritance that violates the structural subtype relationship with the superclass.

**Inheritance in Scala**
```
class Point(xc: Int, yc: Int) {
  val x: Int = xc
  val y: Int = yc
  def move(dx: Int, dy: Int): Point = new Point(x + dx, y + dy)
}
class ColorPoint(u: Int, v: Int, c: String) extends Point(u, v) {
  val color: String = c
  override def move(dx: Int, dy: Int): ColorPoint = new ColorPoint(x + dy, y + dy, color)
}
```

**Dynamic Dispatch**
- The Smalltalk model for method calls:
  - Dynamic dispatch
    - Decision at runtime by the object

**Dynamic dispatch**
- Consider the following class (in Java):
  ```java
  public abstract class StringBuffer {
    public void append(Integer i);
    public void append(Float f);
    public void append(String s);
    public void append(Object o);  
  }
  StringBuffer buf = new StringBuffer();
  ```
Dynamic Dispatch

- What happens in Java?
  - `Integer i = new Integer(10);`
  - `buf.append(i);`
  - `String s = new String("foo");`
  - `buf.append(s);`
  - `Object o = new String("foo2");`
  - `buf.append(o);`

- From this perspective: static dispatch, compiler has already generated the right calls (see next slide)
  - Calls method with the right number of arguments and the most specific parameter type, based on the declared type of the argument

---

Dynamic Dispatch

```
public class Foo {
    public callme() {
    }
}

public class Bar extends Foo {
    public callme() {
    }
}

Foo f = new Bar();
f.callme();
```

---

Dynamic Dispatch

```
Constant pool:
#16 = Class
#17 = Utf8
test/StringBuffer
...
#24 = Methodref
#16.#25
#25 = NameAndType
#26:#27
#26 = Utf8
append
#27 = Utf8
(Ljava/lang/Object;)V
```

```
Code:
aload_1
aload_2
invokevirtual #24
```

---

Single Dispatch

- Bar.callme is called.
- Invokevirtual is dispatched to the actual object and it will invoke its own method (or lowest method in the inheritance chain).

```
Single Dispatch: The question which method is called for
  x.callme(y);
  depends only on the runtime type of x.
```

- Also called a virtual function call.
VTables

- Common implementation:
  - Every class gets a virtual method table at runtime which contains method signatures and addresses to the method implementation
- vtable for Foo:
<table>
<thead>
<tr>
<th>method</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>callme()V</td>
<td>&amp;Foo.callme()V</td>
</tr>
<tr>
<td>dontcallme()V</td>
<td>&amp;Foo.dontcallme()V</td>
</tr>
</tbody>
</table>
- vtable for Bar:
<table>
<thead>
<tr>
<th>method</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>callme()V</td>
<td>&amp;Bar.callme()V</td>
</tr>
<tr>
<td>dontcallme()V</td>
<td>&amp;Foo.dontcallme()V</td>
</tr>
</tbody>
</table>

Multiple Dispatch

- If the call of
  - x.call(a,b,c,...)
  - depends on the runtime type of x and all the arguments a,b,c,... we have multiple dispatch
- Such methods are called Multimethods
- Natively supported by few languages, e.g., CLOS (Common Lisp Object System), Clojure, etc.

(defmethod callme ((x foo) (a integer))
(defmethod callme ((x bar) (a integer))
(defmethod callme ((x bar) (a number))