Run-time binding (or late binding)

- Binding
  - The translation of name into memory address

- Run-time binding
  - The translation is done at run-time
  - also known as
    - late binding
    - dynamic binding
    - virtual invocation

- Polymorphism depends on run-time binding

Possible implementation of run-time binding (polymorphism)

```
// an Account object
obj       // Class<Account>
           Account.getName()
           Account.deposit()

// a SavingAccount object
obj       // Class<SavingAccount>
           Account.getName()
           SavingAccount.deposit()
```

Dynamic binding – under the hood (simplified)

- Compile `obj.deposit()` to `obj.class.dvec[1](obj);`
  - `obj` is a pointer to the object
  - `obj.class` is a pointer to `obj`’s runtime class (getClass())
  - `obj.class.dvec` is a pointer to dispatch vector
  - `obj.class.dvec[1]` is the 2nd slot in the dvec
  - `deposit()` is the second method
  - `obj.class.dvec[1](obj)` passes `obj` as ‘this’ pointer

- If `obj` is an Account, then `Account.deposit()` is called
- If `obj` is a SavingAccount, then `SavingAccount.deposit()` is called

Another example

```java
class A {
    public final void f0() {...};
    public void f1() {...};
    private int a;
}

class B extends A {
    public void f1();
    public void f3();
    protected int b;
}
```

- `f0` is a method that can not be inherited
- `f1()` is overridden by B
- `f2()` has not been overridden
- `f3()` is a new method in B

When to bind?

- void func (Account obj) {
  obj.deposit();
}
- What should the compiler do here?
  - The compiler doesn’t know which concrete object type is referenced by `obj`
  - the method to call can only be known at run time (because of polymorphism)
  - Run-time binding