The Essence of Object-oriented Programming Language

Some core features that are shared by most OO languages

- Inheritance
  - Dynamic dispatch (dynamic binding)
  - Subtype polymorphsim
  - Late binding

Inheritance

A sub-class inherits things from its super-class.
A class is a data structure that can be

- *instantiated* to create new objects ("instances")
- *refined* to create new classes ("subclasses") $\Longrightarrow$ inheritance!

Example

```
class A {
  protected int x = 0;
  int m() { x = x+1; return x; }
  int n() { x = x-1; return x; }
}

class B extends A {
  int o() { x = x*10; return x; }
}
```

An instance of `B` has methods `m`, `n`, and `o`. The first two are inherited from `A`.

Subtype = Subclass

```
class B extends A  int o()  x = x*10; return x;
```

Class B is an *sub-type* of class A.

An object of class(type) B can be considered of class(type) A too, as integer 1 can be considered of real number 1.

Inheritance Enables Dynamic Dispatch

When an operation is invoked on an object, the ensuing behavior depends on the object itself.

Two objects of the *same class(type)* may be implemented internally in *different* ways.

Example (in Java)

```
class A {
  int x = 0;
  int m() { x = x+1; return x; }
  int n() { x = x-1; return x; }
}

class B extends A {
  int m() { x = x+5; return x; }
}

class C extends A {
  int m() { x = x-10; return x; }
}
```

`(new B()).m()` and `(new C()).m()` invoke completely different code!

Inheritance Enables a Form of Polymorphism

"subtype polymorphism": programmers can write one piece of code that operates uniformly on any object of sub-classes

Example

```
// ... class A and subclasses B and C as above...

class D {
  int p (A myA) { return myA.m(); }
}

...

D d = new D();
int z = d.p (new B());
int w = d.p (new C());
```

Late binding

Most OO languages offer an extension of the basic mechanism of classes and inheritance called *late binding*.

Late binding by a special "pseudo-variable" `this`.

Examples

```
class E {
  protected int x = 0;
  int m() { x = x+1; return x; }
  int n() { x = x-1; return this.m(); }
}

class F extends E {
  int m() { x = x+100; return x; }
}
```

- What does `(new E()).n()` return?
- What does `(new F()).n()` return?

Calling "super"

It is sometimes convenient to "re-use" the functionality of an overridden method.

Java provides a mechanism called `super` for this purpose.

Example

```
class E {
  protected int x = 0;
  int m() { x = x+1; return x; }
  int n() { x = x-1; return this.m(); }
}

class G extends E {
  int m() { x = x+100; return super.m(); }
}
```

What does `(new G()).n()` return?