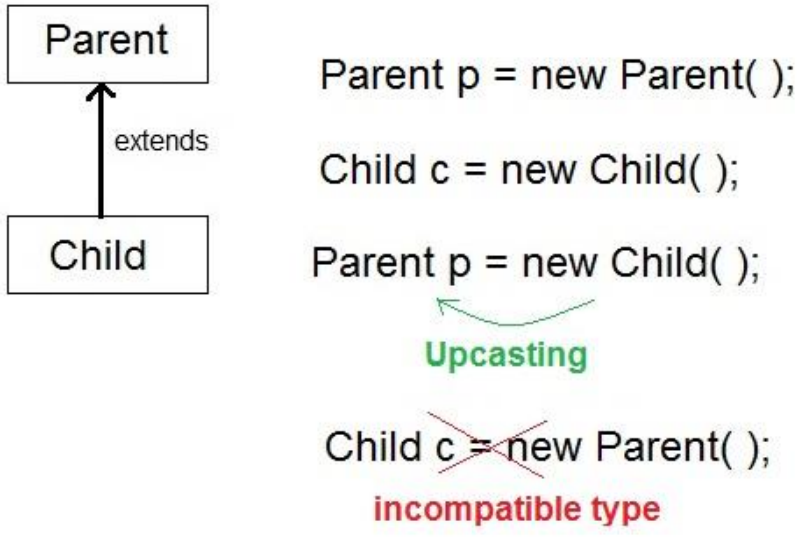
Dynamic Method Invocation

- Method overriding is one of the ways in which Java supports Runtime Polymorphism.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- When an overridden method is called through a superclass reference, Java determines which version (superclass / subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time.

Dynamic Method Invocation

- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.



```
Parent p = new Parent( );

Child c = new Child( );

Parent p = new Child( );

~~Child c = new Parent( );~~
```

Dynamic Method Invocation

- Overriding is a lot more than the namespace convention.
- Overriding is the basis for dynamic method dispatch – a call to an overridden method is resolved at run-time, rather than compile-time.
- Method overriding allows for dynamic method invocation:

1) an overridden method is called through the super-class variable
2) Java determines which version of that method to execute based on the type of the referred object at the time the call occurs
3) when different types of objects are referred, different versions of the overridden method will be called.

Example: Dynamic Invocation

- A super-class A:
- class A {

  void callme() {

  System.out.println("Inside A's callme method");

  }
- }

Example: Dynamic Invocation

- Two sub-classes B and C:

```
• class B extends A {
        void callme() {
        System.out.println("Inside B's callme method");
        }
• }
• class C extends A {
        void callme() {
        System.out.println("Inside C's callme method");
        }
• }
```

- B and C override the A’s callme() method.

Example: Dynamic Invocation

- Overridden method is invoked through the variable of the super-class type.
- Each time, the version of the callme() method executed depends on the type of the object being referred to at the time of the call:

```
• class Dispatch {
• public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
• A r;
• r = a; r.callme();
• r = b; r.callme();
• r = c; r.callme();
• } }
```

Uses of final

- The final keyword has three uses:

1) declare a variable which value cannot change after initialization
2) declare a method which cannot be overridden in sub-classes
3) declare a class which cannot have any sub-classes

Uses of final

- Java final variable
- When a variable is declared with final keyword, its value can’t be modified, essentially, a constant. This also means that you must initialize a final variable.
- We must initialize a final variable, otherwise compiler will throw compile-time error.A final variable can only be initialized once, either via an initializer or an assignment statement.
- class FinalVariable {

      final int var = 50;

      var = 60 //This line would give an error
- }

Preventing Overriding with final

- A method declared final cannot be overridden in any sub-class:

```
class A {
        final void meth() {
                System.out.println("This is a final method.");
        }
}
```

- This class declaration is illegal:

```
class B extends A {
        void meth() {
                System.out.println("Illegal!");
        }
}
```

final and Early Binding

- Two types of method invocation:

1) early binding – method call is decided at compile-time
2) late binding – method call is decided at run-time

- By default, method calls are resolved at run-time.
- As a final method cannot be overridden, their invocations are resolved at compile-time.
- This is one way to improve performance of a method call.

Preventing Inheritance with final

- A class declared final cannot be inherited – has no sub-classes.
- final class A { … }
- This class declaration is considered illegal:
- class B extends A { … }
- Declaring a class final implicitly declares all its methods final.
- It is illegal to declare a class as both abstract and final.