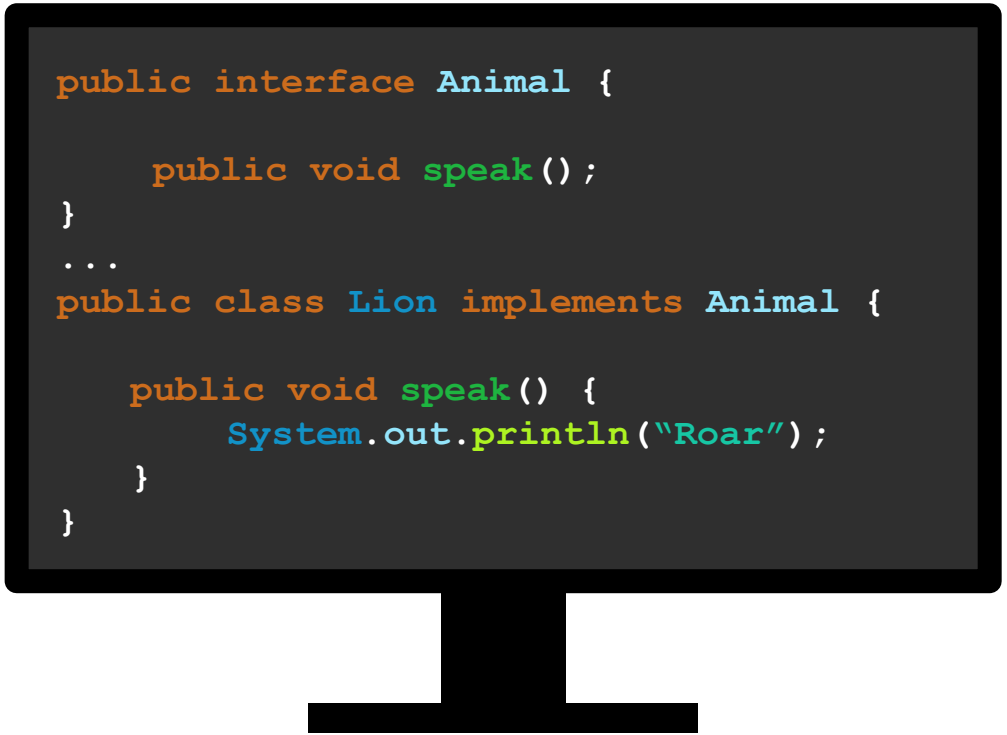
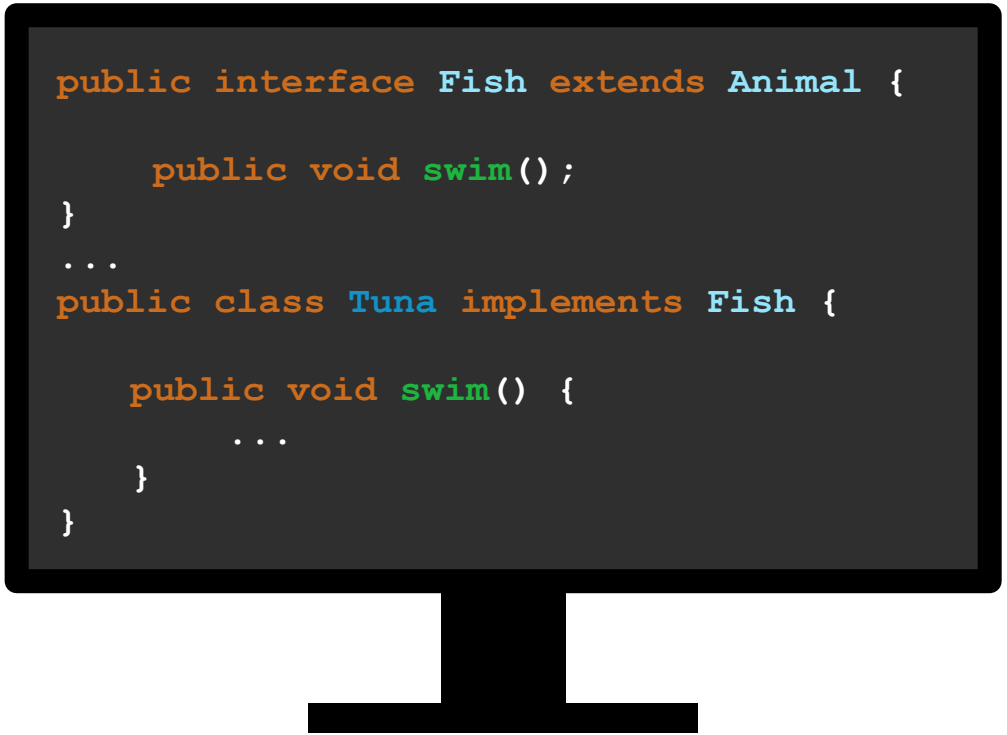
Interface

- **Interface** is a contract
- Contains **method signatures** (methods without implementation) and **static constants** (final static)
- Can only be implemented by classes and extended by other interfaces
- Used as a “template” for how a class should be structured



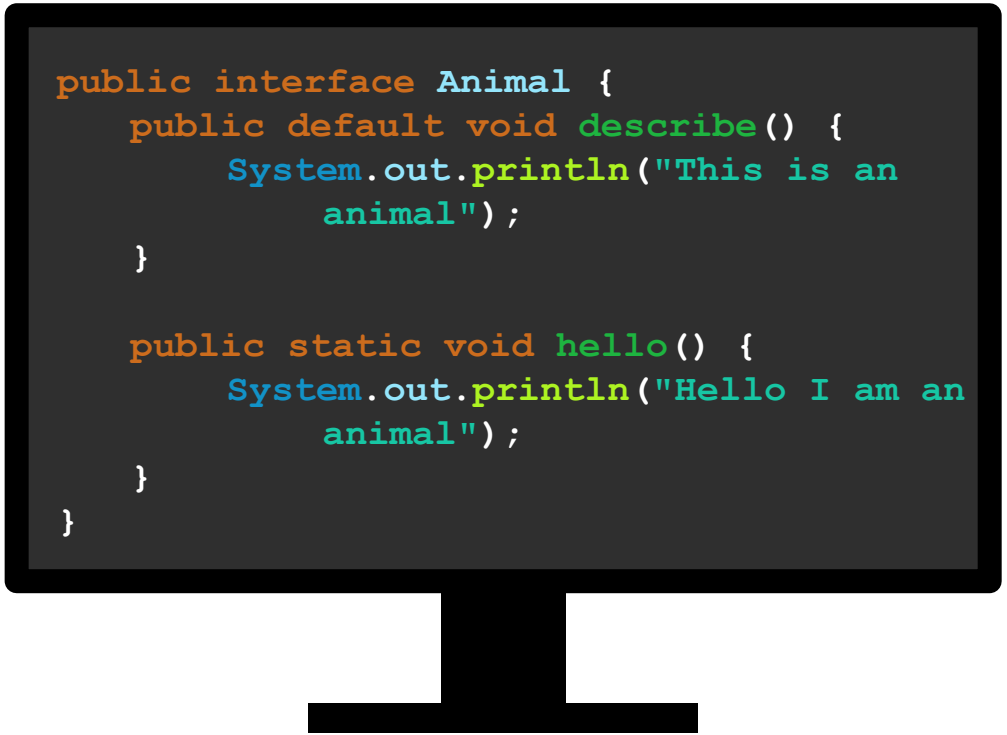
Interface Inheritance

- A class inheriting from an interface uses the keyword **implements**
- An interface inheriting from another interface uses the keyword **extends**



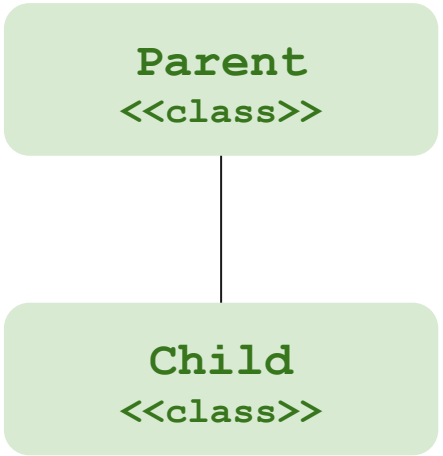
Java 8 Interfaces

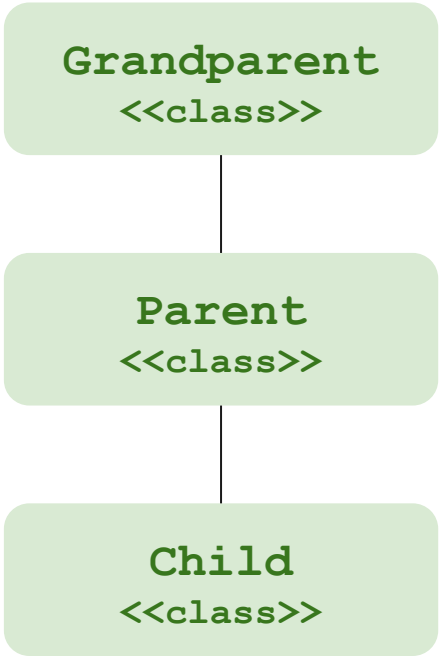
- With Java 8, interfaces can have default and static methods
- **Default methods** allow for a default method for an interface, can be overwritten
- **Static methods** are methods that belong to the interface



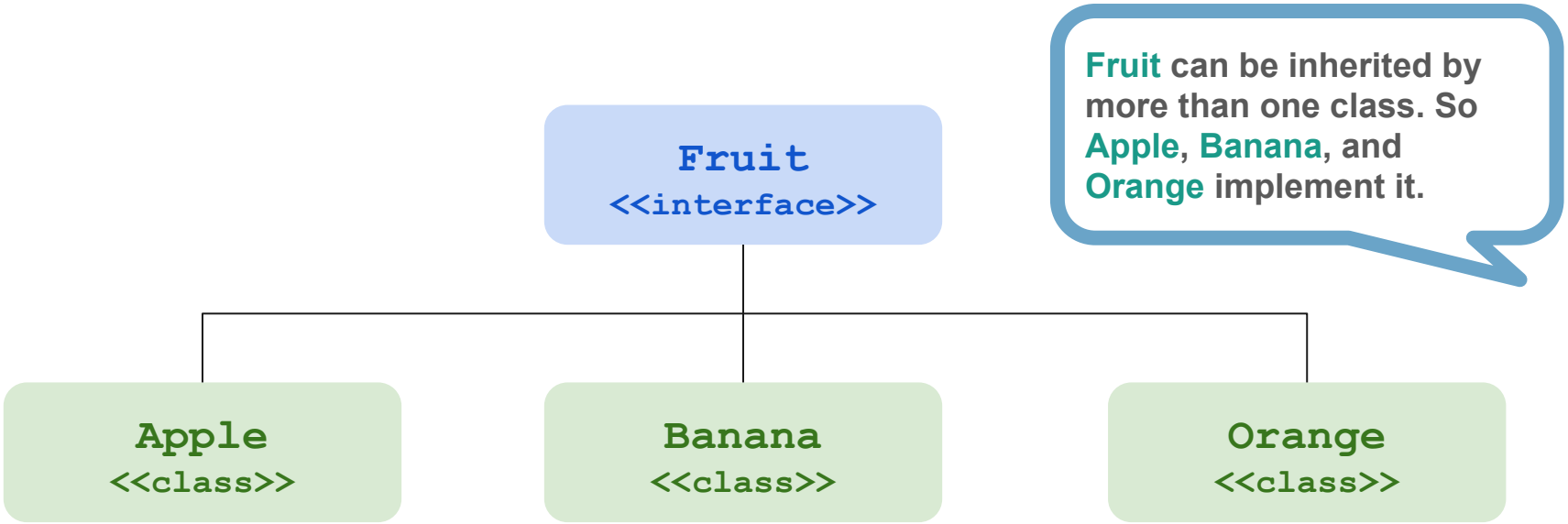
Single and Multilevel Inheritance

Child can inherit from Parent on left, but the multilevel on right allows Child to inherit from both Parent and Grandparent.

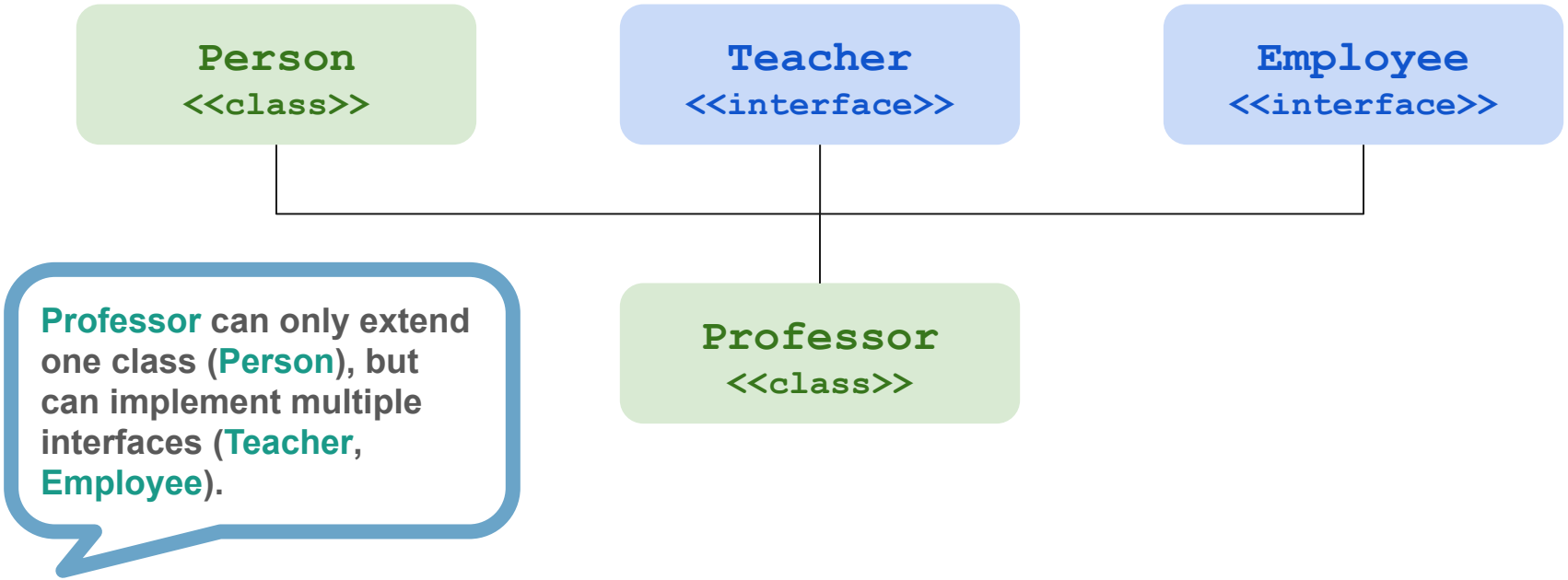




Hierarchical Inheritance



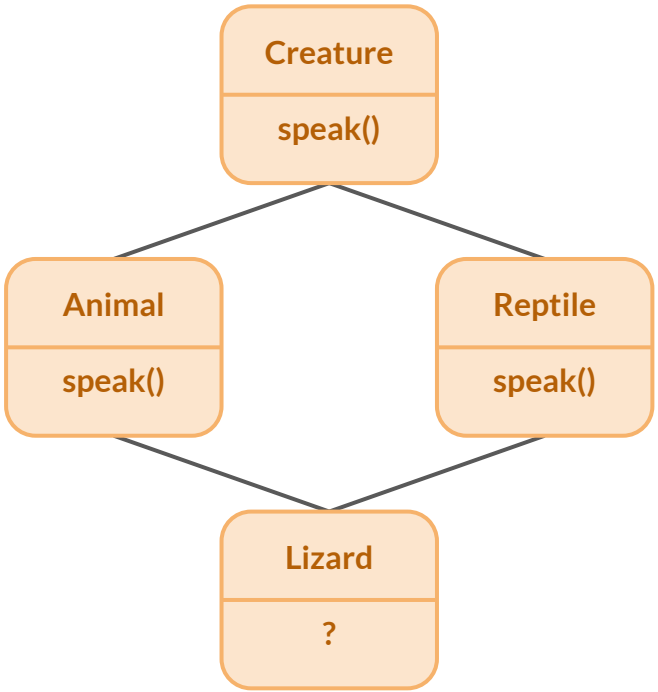
Multiple Inheritance



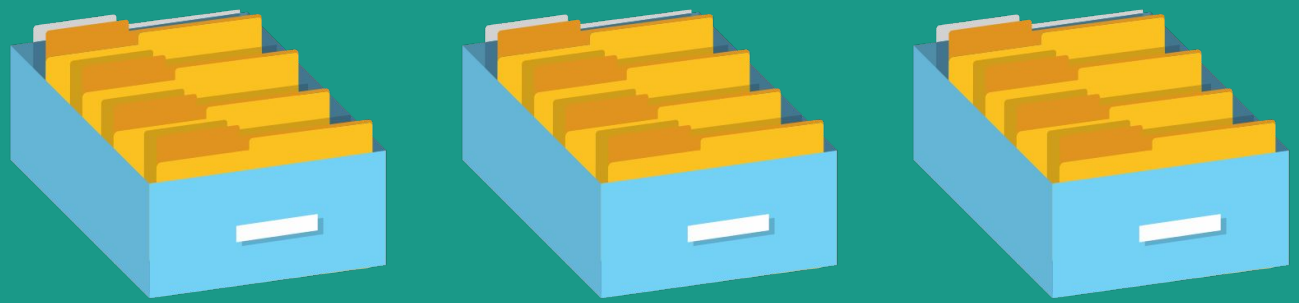
Hybrid Inheritance

Diamond Problem

Will Lizard inherit the method speak from Animal or Reptile? Or could it possibly inherit it from Creature?



Packages and Imports

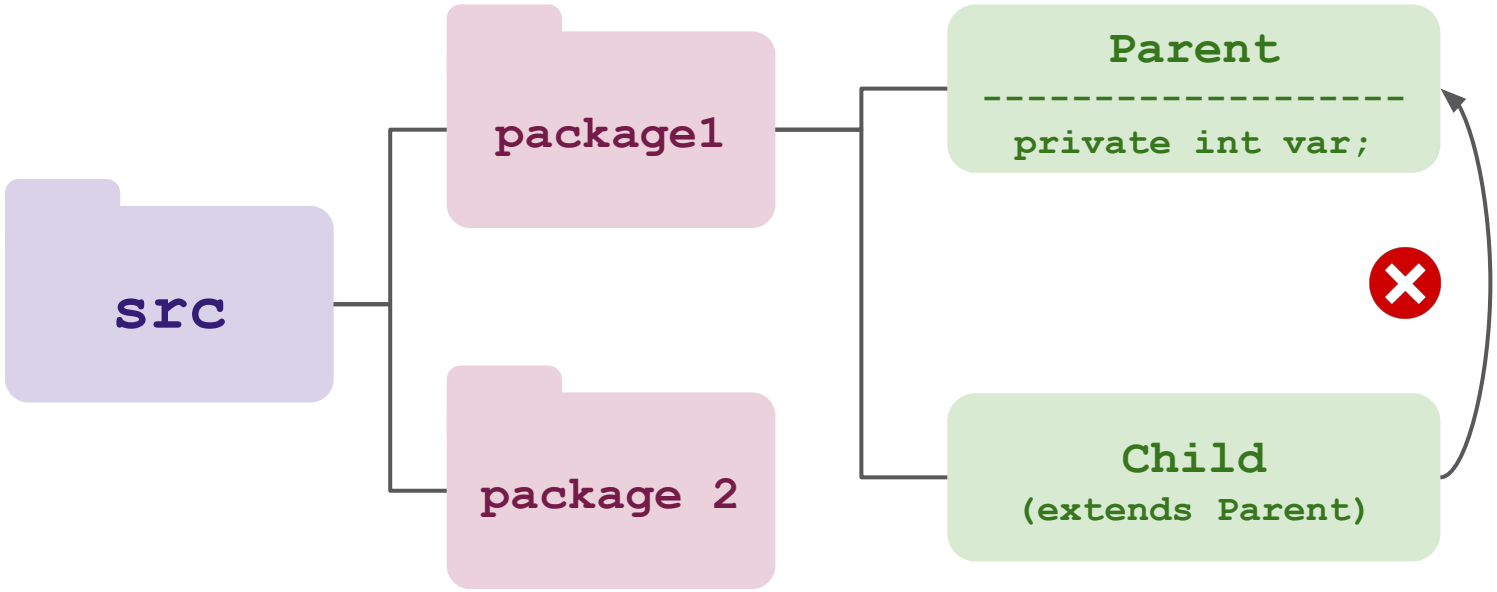


Packages and Imports

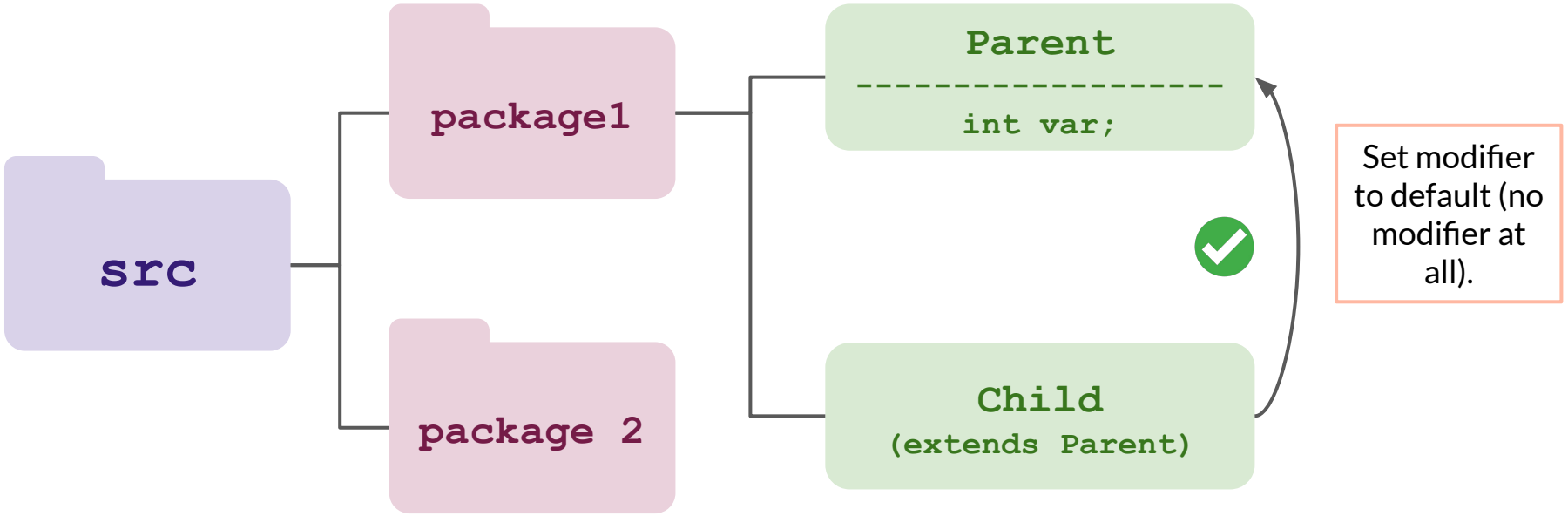
- **Packages** - Java mechanism to organize classes
  - Naming Conventions:
    - i.e. - com.cognixia.jump.corejava
    - OrganizationType.CompanyName.OrganizationTopic
- **Imports** - Java needs to know what libraries to reference

  to use certain Classes
  - I.e. - java.lang, java.util

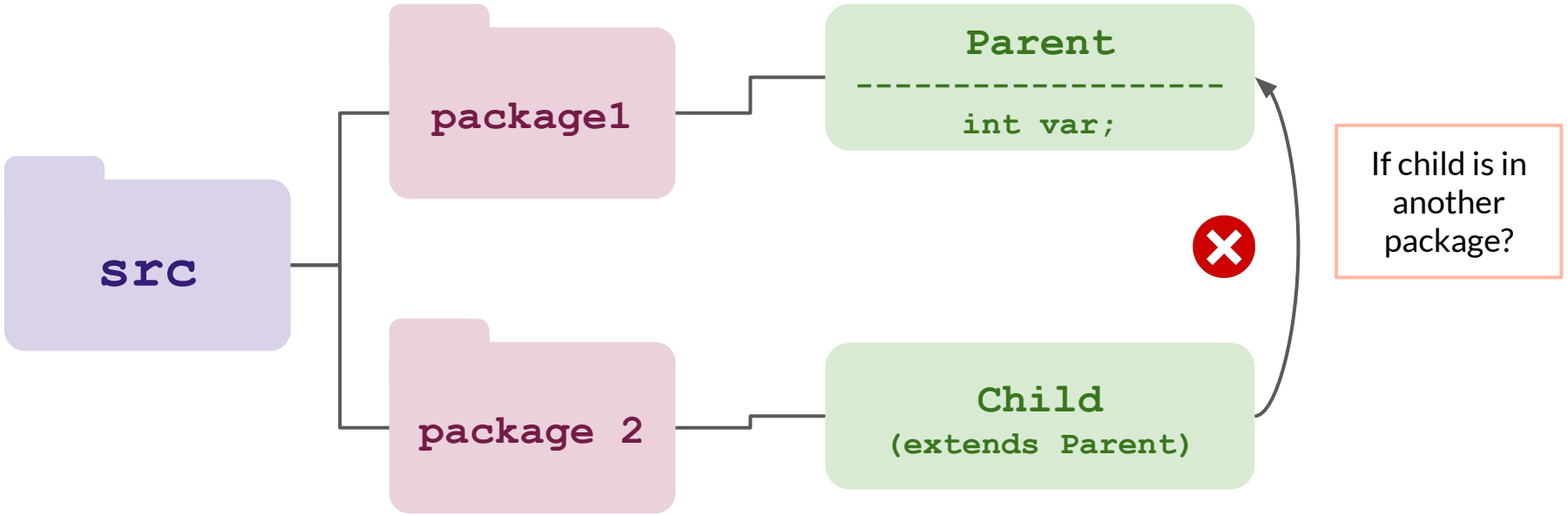
Access Modifiers: Default



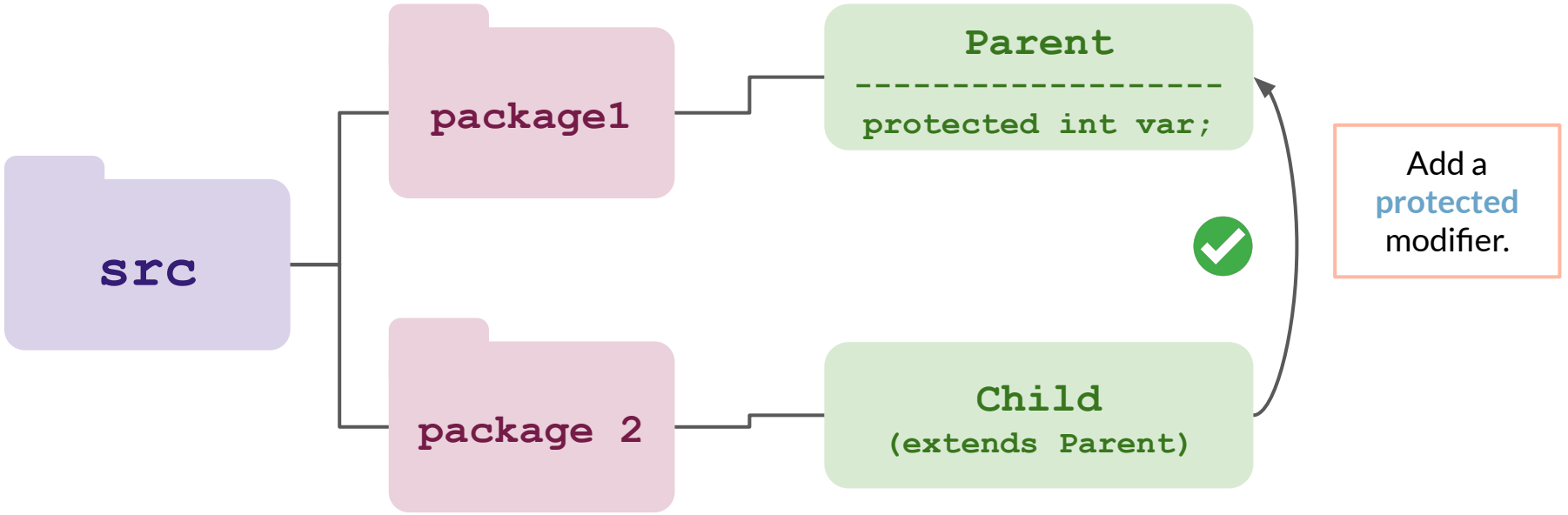
Access Modifiers: Default



Access Modifiers: Protected



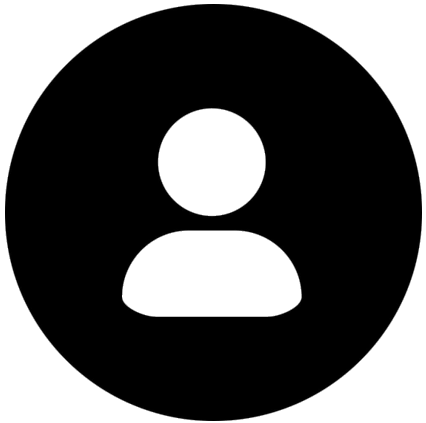
Access Modifiers: Protected



Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	Allowed	Allowed	Allowed	Allowed
Protected	Allowed	Allowed	Allowed	Denied
Default	Allowed	Allowed	Denied	Denied
Private	Allowed	Denied	Denied	Denied

Collections and Generics

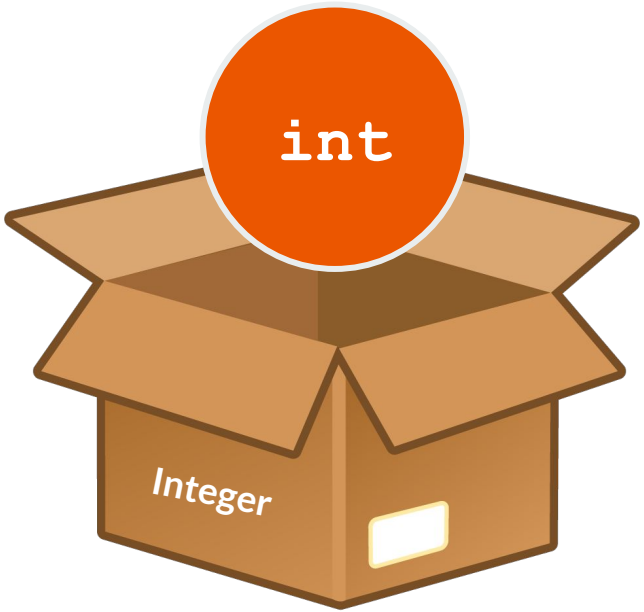


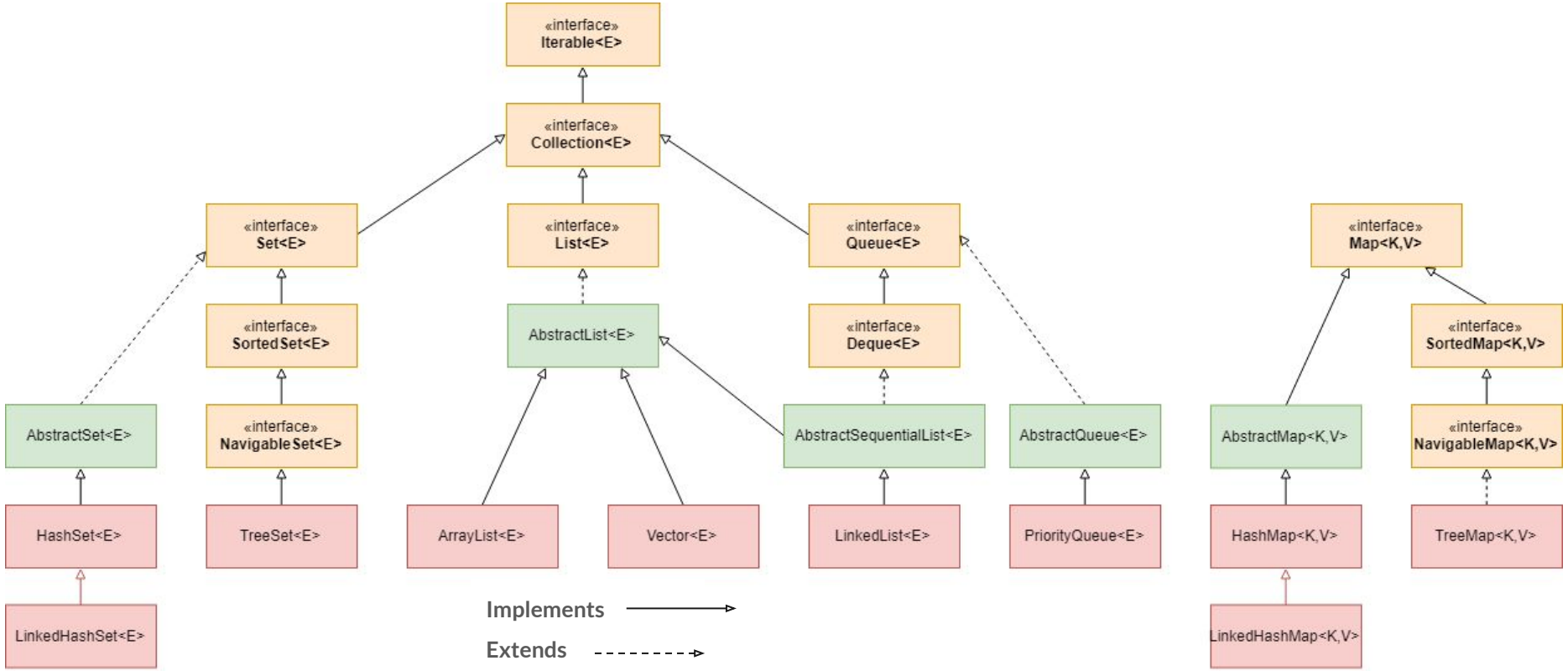
Primitive v Autoboxed

- **Java** is NOT a pure **OOP** language
  - has **primitives**
- **Objects** are required in many applications for Java
- **Wrapper Classes** - objects that wrap around the primitive type
- Can use **Collections** - Collections cannot store primitives on their own

```
double dubs = 5.0;
int num = (int) dubs;

// passes int to boxed type
Integer boxed = num;
```





Collections Framework - provides a set of classes and interfaces that can store and manipulate groups of objects

List

- **List** supports methods to maintain a collection of objects as a linear list
  - $L = (l_0, l_1, l_2, \ldots, l_N)$
- No limit to objects that can be added
- Size is dynamic
- Classes that implement List:
  - **ArrayList** - array used to manage data
  - **LinkedList** - stores objects using linked-node representation
  - **Vector** - like ArrayList, but synchronized

```
List<Tree> trees = new ArrayList<Tree>();
trees.add(new Tree("Oak"));
trees.add(new Tree("Pine"));
trees.add(new Tree("Maple"));
trees.remove(0);
trees.add(new Tree("Palm"));

for( Tree tree : trees ) {
    System.out.println(tree);
}
// prints: Pine Tree, Maple Tree, Palm
//         Tree
```

Frequently Used List Methods

`boolean add( E e )`	Add an element to the list (must be an object)
`int size()`	Returns the number of elements in the list
`E get( int index )`	Returns the element at the index given
`E remove( int index )`	Removes element at index given, returns the element removed
`boolean remove( E e )`	Removes the element passed, will return true or false if element given was found and removed
`void clear()`	Clears list (removes all elements)
`boolean isEmpty()`	Returns true or false if list is empty

Homogeneous vs Heterogeneous Collections

- Homogeneous collections include objects of a single type

```
List<Monster> monsters = new ArrayList<Monster>();
monsters.add(new Monster());
monsters.add(new Monster());
...
```

- Heterogeneous collections include objects of a variety of types (derived from the same base class)

```
List<Monster> monsters = new ArrayList<Monster>();
monsters.add(new Vampire()); // Vampire and Mummy inherit
monsters.add(new Mummy());   // from Monster
...
```

Set

- **Sets** are an unordered collection of objects with no duplicates
- Classes that implement Set:
  - **TreeSet** - sorts any element added to it
  - **HashSet** - implemented using hash table and is faster than a TreeSet at access data

```
Set<String> colors = new
                  TreeSet<String>();
colors.add("red");
colors.add("blue");
colors.add("red");
colors.add("green");
colors.add("blue");
colors.add("yellow");

System.out.println(colors);
// prints: [blue, green, red, yellow]
```

Iterator

- An **Iterator** is an object used with collections to provide sequential access to that collection's elements
- Can impose ordering on elements if none
- Can alternately use ***for-each loop*** for certain situations

Map

- Maps contain methods that maintain a collection of objects with ***key-value pairs*** called map entries
- Classes that implement Map:
  - TreeMap - sorts pairs within it
  - HashMap - not sorted, uses a hash table to access pairs by their key

```
Map<String,Integer> coins = new
        TreeMap<String,Integer>();
coins.put("penny", 1);
coins.put("nickel", 5);
coins.put("dime", 10);
coins.put("quarter", 25);

System.out.println(coins);
// prints: {dime=10, nickel=5, penny=1,
//          quarter=25}
```

Hashtable vs HashMap vs ConcurrentHashMap

Hashtable	HashMap	ConcurrentHashMap
➔ ***Synchronized*** ➔ ***No null keys or values*** can be passed to it ➔ Puts ***lock on whole map*** so all methods and access to data synchronized	➔ ***Not synchronized***, not thread safe ➔ Allows for ***one null key*** and ***multiple null values***	➔ ***Synchronized*** ➔ ***No null keys or values*** can be passed to it ➔ Doesn't lock whole map, ***locks it in segments***, data that needs to be updated, will lock only segment its in

Generics

- Introduced in Java 5
- Before, could store any type of objects in a collection
- Generics create type safe collections
- Advantages:
  - *Type safe*
  - *Type casting not required*
  - *Compile-time checking*