Interfaces
& Sub-types

Weiss sec. 4.4

Scenario

- Instructor says:

  “Implement a class IntegerMath with two methods *pow* and *fact* with the following signatures:

  public static int power(int a, int b);

  public static int factorial(int n);

  that compute …”

Scenario

- Instructor says:

  "Implement a class IntegerMath with two methods *pow* and *fact* with the following signatures:

  static int power(int a, int b);

  static int fact(int n);

  that compute …"

- Result:
  - Student asks "Did you mean *pow* or *power*?"
  - Student turns in:

```
public class IntigerMath {
    public static double pow(double a, int n) { …}
    public static int factorial(int x) { …}
}
```

  - Student's code compiles fine, but Grader's test program won't compile!

Software Engineering

- How to write a large program (say, 1 million lines):
  - Find a smart & productive person:
    - 1000 lines/day x 365 days/year x 2.7 years = 1 million lines
  - Split program into many small units:
    - Each unit assigned to one person or team
    - Each unit has a specification
      - defines what it is supposed to do (and maybe how)
    - Each unit has an interface
      - defines “what it looks like” from the outside
    - Assigned person/team writes implementation
      - must follow specification
      - must implement the interface
    - Then someone puts it all together
      - hope it all works!

Example 2: Collections

- Specification:
  - Want a class that stores a collection of objects
  - It should have methods for adding/removing objects, finding objects, etc.
- Interface:
  - The objects should be referred to as Collection instances
  - Methods have these signatures:

    public void add(Object e);
    public void remove(Object e);
- Implementation:
  - Farmed out to student(s)
  - Create class, copy signatures, fill in bodies, add helper methods if needed, add static/instance variables if needed, etc.

Compiler

- What can the compiler do to help?
- Implementation satisfies specification?
  - Can’t be checked by compiler (yet)
- Implementation satisfied interface?
  - Can be checked by compiler:
  - Just compare method signatures in implementation against those in the interface definition

Multiple Implementations

- Why?
  - Lots of students
- Why, in the real world?
  - Competing groups
  - Easy to compare different implementations – just plug a new implementation into the program, see how it works
  - implementation evolves/improves over time
- Multiple implementations, one interface: How?
  - Give interface a name: interface Collection
  - Give each implementation a different name:

```
class MSCllctn implements Collection // Microsoft
class AppleBag implements Collection // Apple
class GnuLinux implements Collection // FSF
```

Interfaces in Java

- Elements:
  - interface name + method signatures + constants
  - other classes will implement the methods
- Caveats:
  - all instance methods implicitly “public” and non-static
  - all instance fields implicitly “public static final”
  - **no static methods allowed**
  - no non-final or non-static fields allowed
  - **can’t instantiate directly (b/c it has no body!)**
- Why no static methods?
  - Java interfaces are concerned with interface to an *object*, not to a “bag-of-methods” style class

Collections: One Scenario

- Your boss says:
  "We need a collection that can do the following:
  - add a new object to the collection
  - remove a given object from the collection
  - etc.

  and I don't care how you implement it"
- Someone OKs the following interface with the boss:

```
public interface Collection {
    void add(Object e);
    void remove(Object e);
    boolean contains(Object e);
    void clear();
    // ...
}
```

Collections: One Scenario

- You are given interface (& specification too)

```
public interface Collection {
    void add(Object e);
    void remove(Object e);
    boolean contains(Object e);
    void clear();
    // ...
}
```

- You write:

```
public class LinkedList implements Collection {
  private ListCell head;  // the list contents

  public void clear() { … }
  public void add(Object e) { … }
  public void remove(Object e) { … }
  public boolean contains(Object e) { … }

  // and constructors
  // and helpers: insertHead, search(), getHead(), …
}
```

Multiple Interfaces

- Scenario: Your class can do more than just fulfill the Collection interface.
  - e.g., can also be reversed, saved on disk, etc.

class LinkedList implements Container, Reversible, Comparable, Storable { … }

  - Just need to implement *all* of the required methods

Generic Programming

- Software engineering: specify interface → create a class that implements it
- Generic programming: create lots of similar classes → specify an interface that works with all of them
- Why?
  - Lets us write “generic code”

Example: Print a Linked List

```
 // print a LinkedList
public static void printAll(LinkedList t) {
  for (int i = 0; i < t.size(); i++) {
    Object e = t.get(i);
    System.out.println(i+" : "+e);
  }
}
```

Example: Print other Collections

```
// print a LinkedList
public static void printAll(LinkedList t) {
  for (int i = 0; i < t.size(); i++) {
    Object e = t.get(i);
    System.out.println(i+" : "+e);
  }
}
```

```
// print a Doubly-linked list
public static void printAll(DLinkedList t) {
  for (int i = 0; i < t.size(); i++) {
    Object e = t.get(i);
    System.out.println(i+" : "+e);
  }
}
```

```
// print an ArrayList
public static void printAll(ArrayList t) {
  for (int i = 0; i < t.size(); i++) {
    Object e = t.get(i);
    System.out.println(i+" : "+e);
  }
}
```

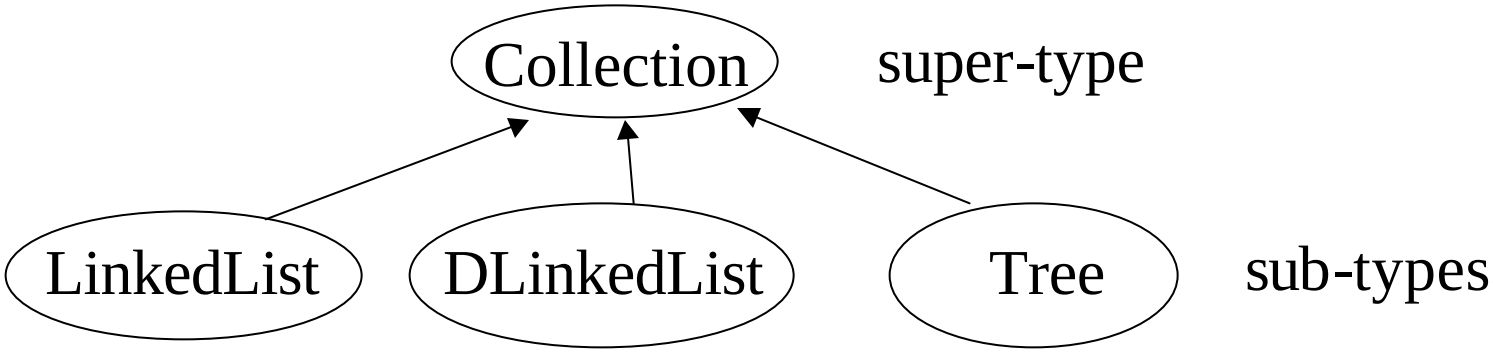
```
// print a Tree
public static void printAll(Tree t) {
  for (int i = 0; i < t.size(); i++) {
    Object e = t.get(i);
    System.out.println(i+" : "+e);
  }
}
```

Ideal: Generic “printAll”

```
 // print anything that implements Collection interface
public static void printAll(Collection t) {
  for (int i = 0; i < t.size(); i++) {
    Object e = t.get(i);
    System.out.println(i+" : "+e);
  }
}
```

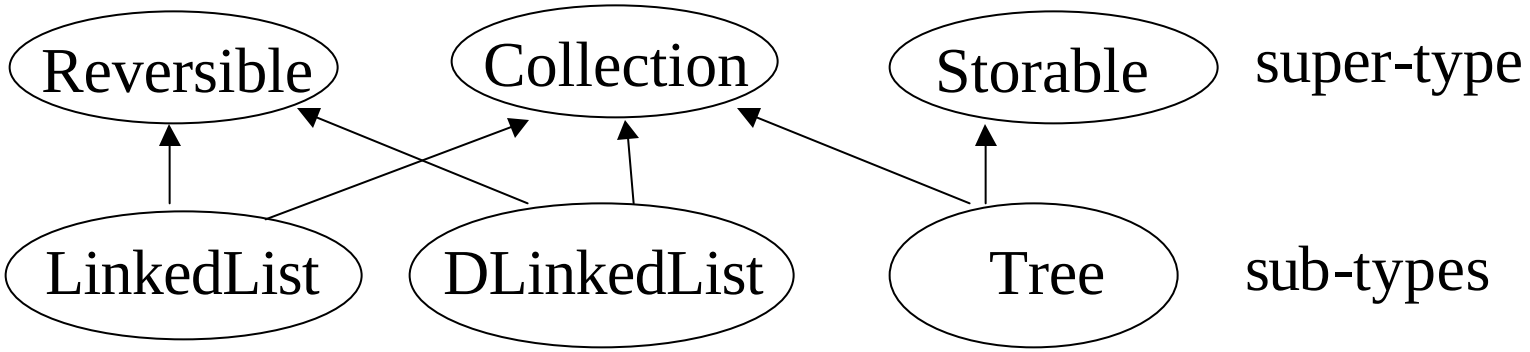
- All we need is a certain kind of object *t*
  - Must have a method called *size*( ) returning *int*
  - Must have a method called *get*(int i) returning *Object*
  - Anything that implements collection has these

Interfaces as Types



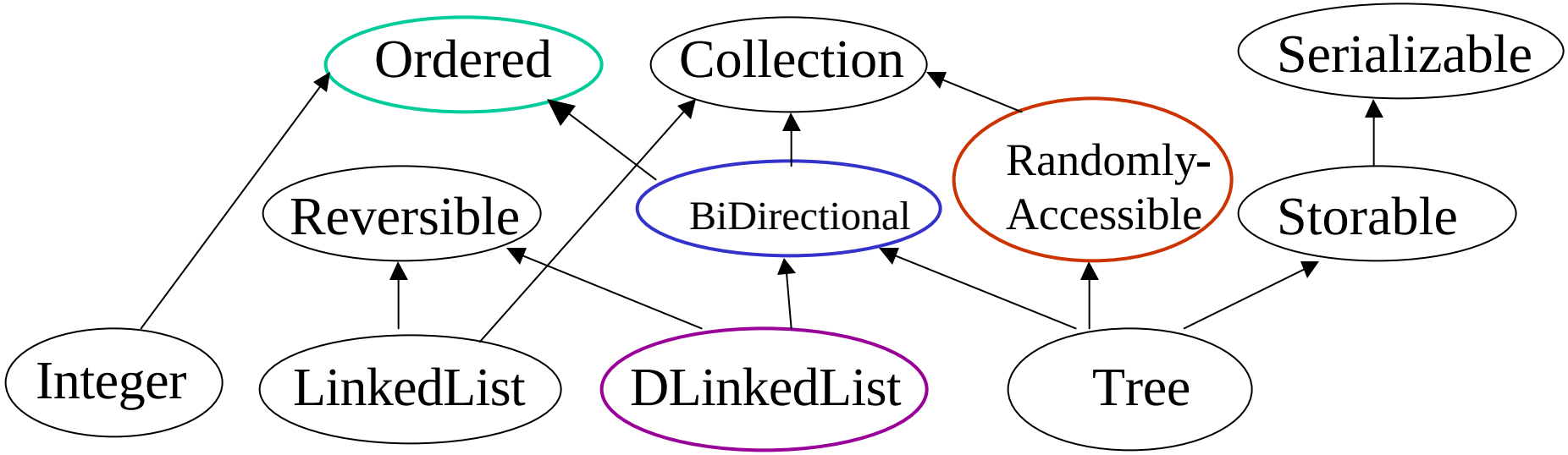
- Name of interface can be used as a variable type:
  - e.g., Collection t1; Collection t2
- Visualize relationship as a hierarchy (tree?)

Multiple Interfaces



- Name of interface can be used as a variable type:
  - e.g., Collection t1; Collection t2;
- A class can have many super-types
- An interface can have many sub-types

Super-Interfaces




```
interface Ordered {
    boolean comesBefore(Object o);
    boolean comesAfter(Object o)
}
```

```
interface Traversable extends Collection, Ordered
{
    Object[] traverseForward();
    Object[] traverseBackward();
}
```

```
interface RandomlyAccessible extends
Collection {
    Object getRandomElement();
}
```

```
class DLinkedList implements
Reversible, BiDirectional {
    …
}
```

Types, but no instantiation

- Can’t instantiate an interface directly:
  - Reversible r = new Reversible(…); // no such constructor
  - There is no “body” (implementation) for Reversible itself
- So what can we do with “Reversible s”?
  - Call methods defined in interface Reversible:
    - e.g we can reverse it using s.reverse();
  - But nothing else
    - no constructors in interface
- So why bother having a variable “Reversible r”?
  - How do we get an instance of Reversible?
  - Want to do: Reversible s = new LinkedList(…);

Type Checking: Assignments

- x = y; // is this okay?
- Without sub-typing, it is easy:
  - String s = new Integer(3); // illegal: String != Integer
  - Rule: LHS type must be same as RHS type
- With sub-typing, it is complicated:
  - LinkedList p = new LinkedList(…); // okay: LHS=RHS
  - Reversible r = p; // okay: RHS is LinkedList, which implements Reversible
  - Tree t = r; // not okay: RHS is Reversible, which may not be a Tree
- Think about “is a” relation versus “may be a”:
  - a LinkedList object “is a” Collection (upward in heir.)
  - a Collection object “may be a” Tree (downward in heir.)

Apparent vs. Actual Types

- Apparent type: what the variable declaration said
  - Reversible r; ← says that r will be a Reversible object
- Actual type: what ended up getting assigned
  - r = new LinkedList(); ← r is now a LinkedList object
- Why bother?

  Reversible r = new LinkedList(…);

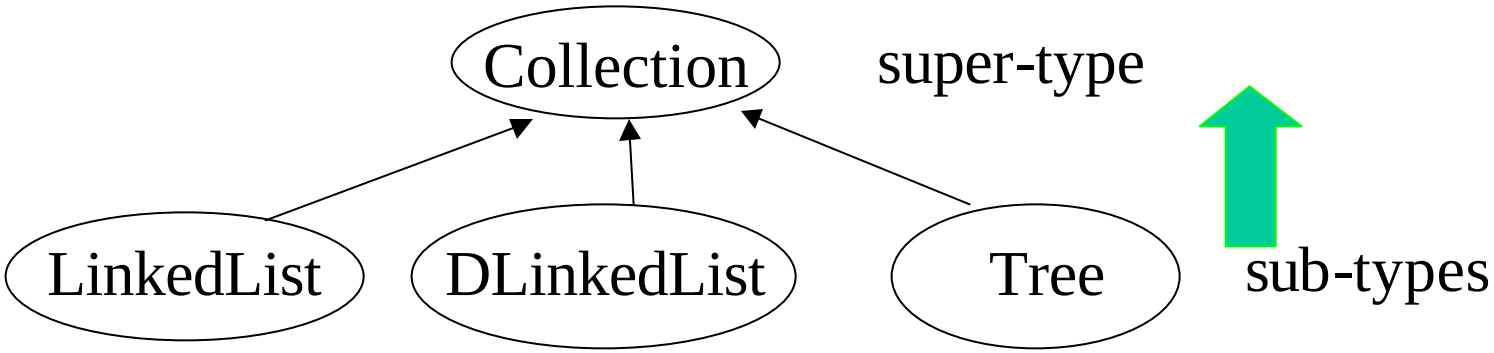
  …

  LinkedList t = r; // is this okay?
- Apparent type: can tell by looking at declaration
- Actual type: have to trace through code at run-time

Static Type Checking

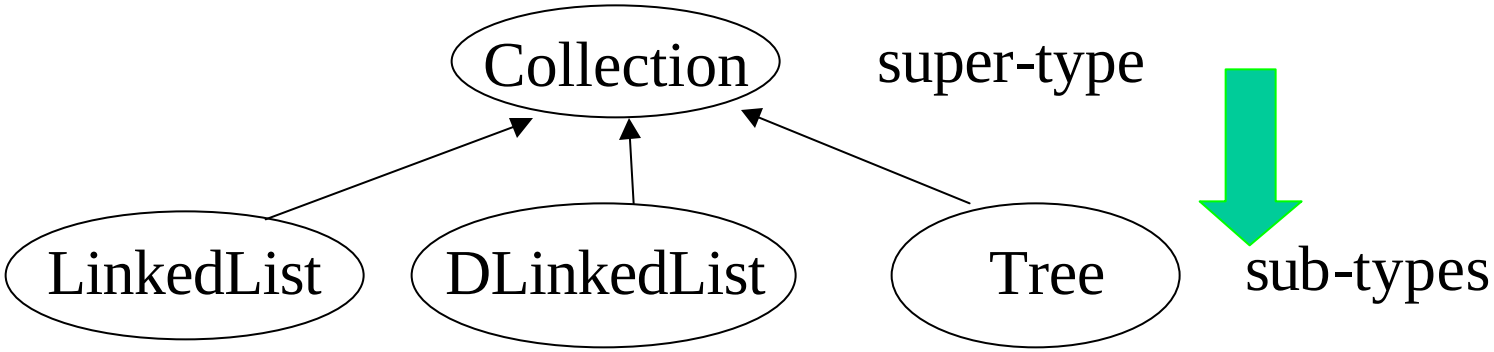
- Java does *static type checking*. In other words:
  - All assignments checked when you compile program
  - If assignment **always** okay, then type check passes
  - If assignment **might not be** okay, then type check fails
  - Uses **apparent** types of variables
- Some other languages to *dynamic type checking*. I.e:
  - All assignments checked when you run your program
  - If assignment **is** okay, then type check passes
  - If assignment **is not** okay, then type check fails
  - Uses **actual** types of variables
- Tradeoffs:
  - dynamic is slow, error-prone, but “quick-and-dirty”
  - static is zero-cost, identifies potential bugs, but sometimes inconvenient (e.g., reports “type check fails” too often)

Up-Casting



- Going from sub-type to super-type
- Apparent type of RHS is sub-type of apparent type of LHS
- Always okay; Type check passes
- Compiler can check this easily – look at declarations
- e.g. Tree t = …; // apparent type Tree
  Collection c = …; // apparent type Collection
  …
  c = t; // always okay; so passes

Down-Casting



- Going from super-type to sub-type
- Apparent type of RHS is super-type of apparent type of LHS
- Sometimes okay; Type check fails
- Compiler can check this easily – look at declarations
- e.g. Tree t = …; // apparent type Tree
  Collection c = …; // apparent type Collection
  …
  c = t; // always okay; so passes

Up-Casting to Generic Code

```
interface Container {
    Object get(int i);
    int size();
}
```

```
class LinkedList implements Container {
  ... get(int i) ... size() ... reverse() ...
}
```

```
class Tree implements Container {
    ... getRoot() ... size() ... get(int i) ...
}
```

Up-Casting to Generic Code

```
class ThirdParty {
   void main(String []args) {
        LinkedList p = …
        Tree t = …
        printAll(p);
        printAll(t);
   }
}
```

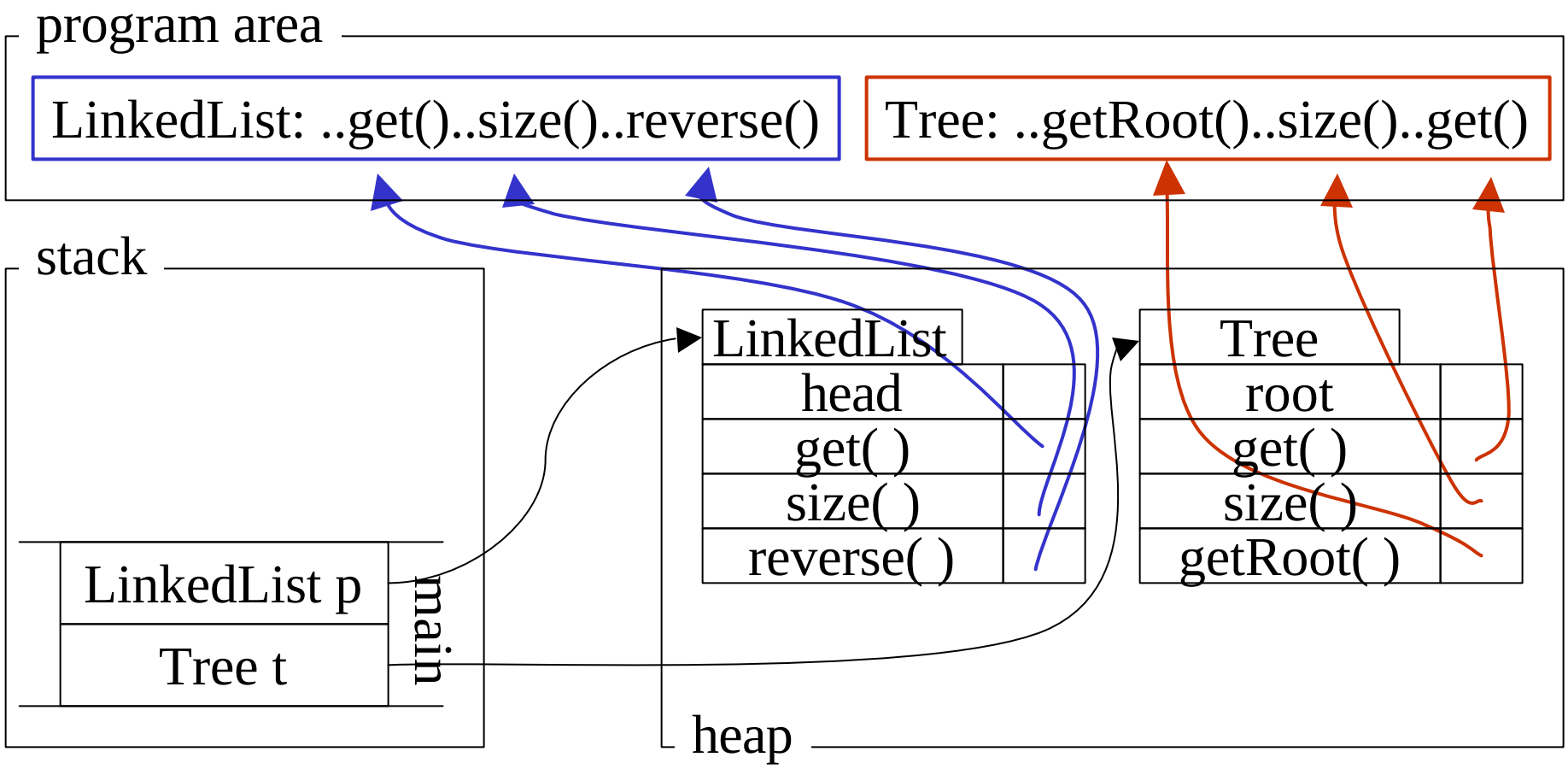
```
public static void printAll(Collection c) {
  for (int i = 0; i < c.size(); i++) {
    Object e = c.get(i);
    System.out.println(i+" : "+e);
  }
}
```

- Question: which *size* method is called from within printAll( ) ?

Dynamic Binding

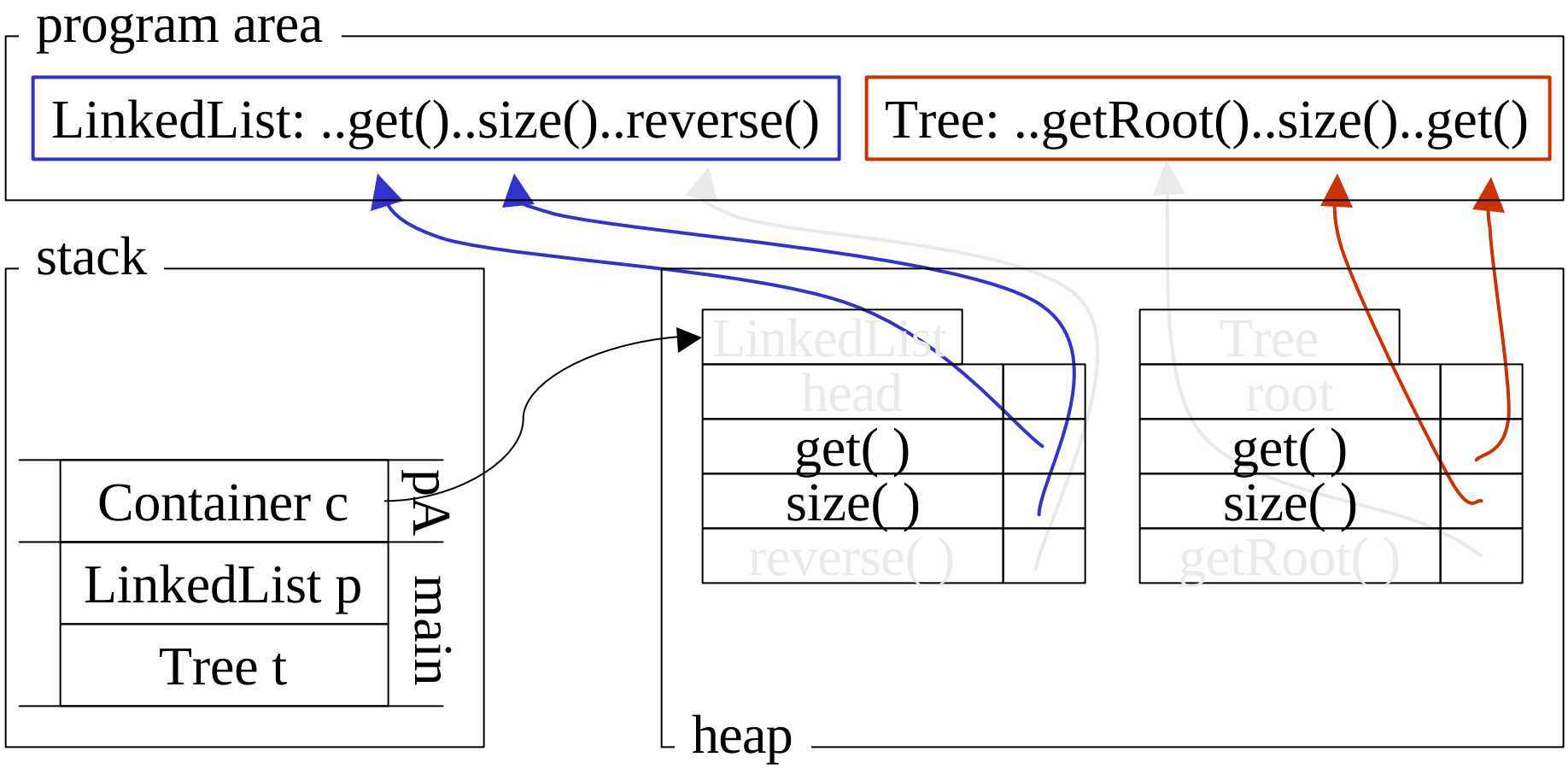
- Answer: Depends on the *actual type* of the object.
  - use size() from Tree? Sometimes
  - use size() from LinkedList? Sometimes
  - use size() from Container? No such thing
- How can compiler tell which?
  - main( ) : try to look at variable declarations – might work
  - printAll( ) : variable declaration does not help

Dynamic Binding



In this example, main( ) sees both objects as they actually are

Dynamic Binding



printAll sees only Container methods

During execution: follow reference from *c*, find method you want, follow reference to find code

Summary

- Interfaces have two main uses:
  - Software Engineering:
    - Good fences make good neighbors
  - Sub-typing:
    - Interface is super-type; implementation is sub-type
    - Use to write more “generic” code
- Sub-typing:
  - Think: “is-a” relationship
  - Several ways to do this in Java (interfaces are one way)
  - Up-casting: LHS is super-type of RHS – always okay
  - Down-casting: LHS is sub-type of RHS – not always okay
- Dynamic binding: code to run is found at run-time
- Static type checking: compiler checks all assignments