Structured Programming

COMP1110/6710

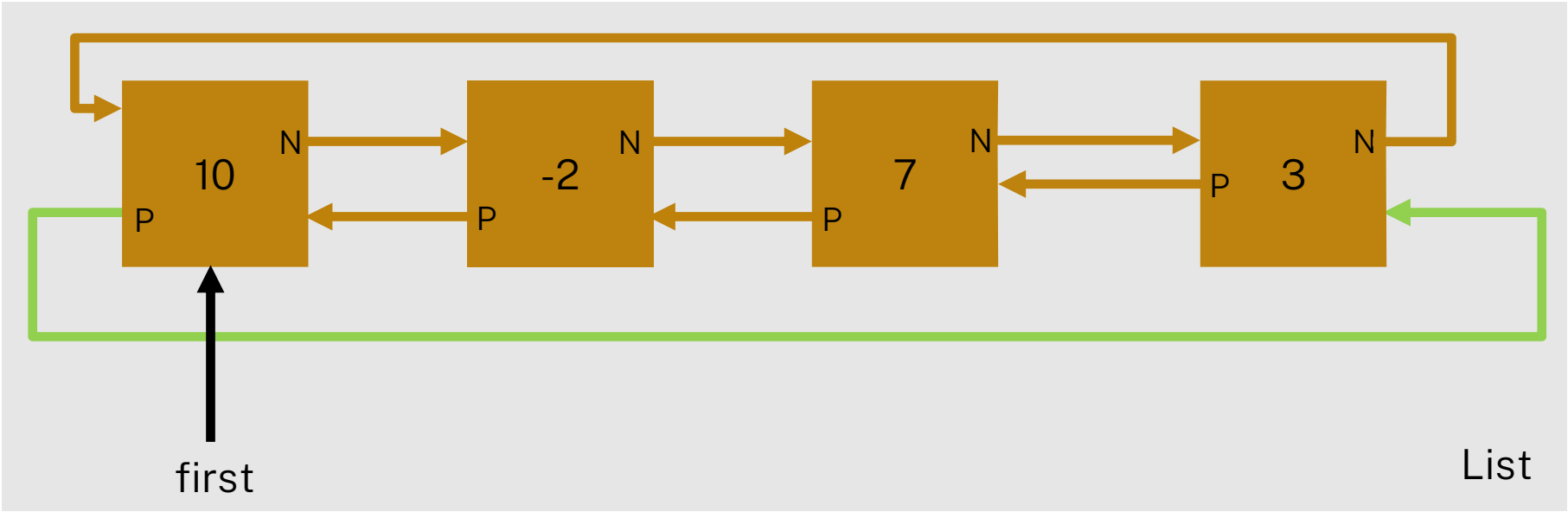
Australian National University

Image Courtesy NASA/JPL-Caltech.

A Few More Things on Iterators

Australian National University

Recall: Doubly-Linked List



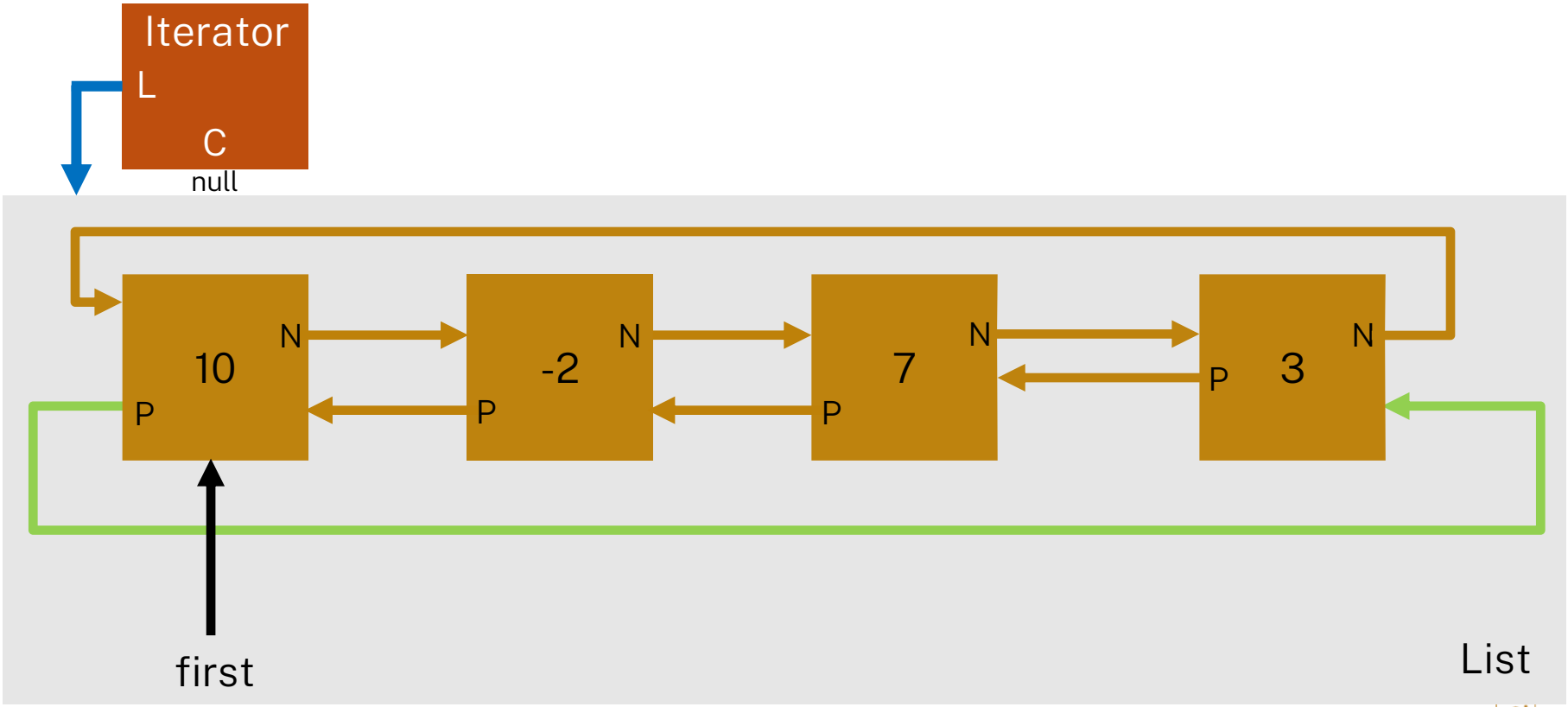
Double-Linked List: Iteration

```
for(int i = 0; i < list.size(); i++) {
    var element = list.get(i);
    …
}
```

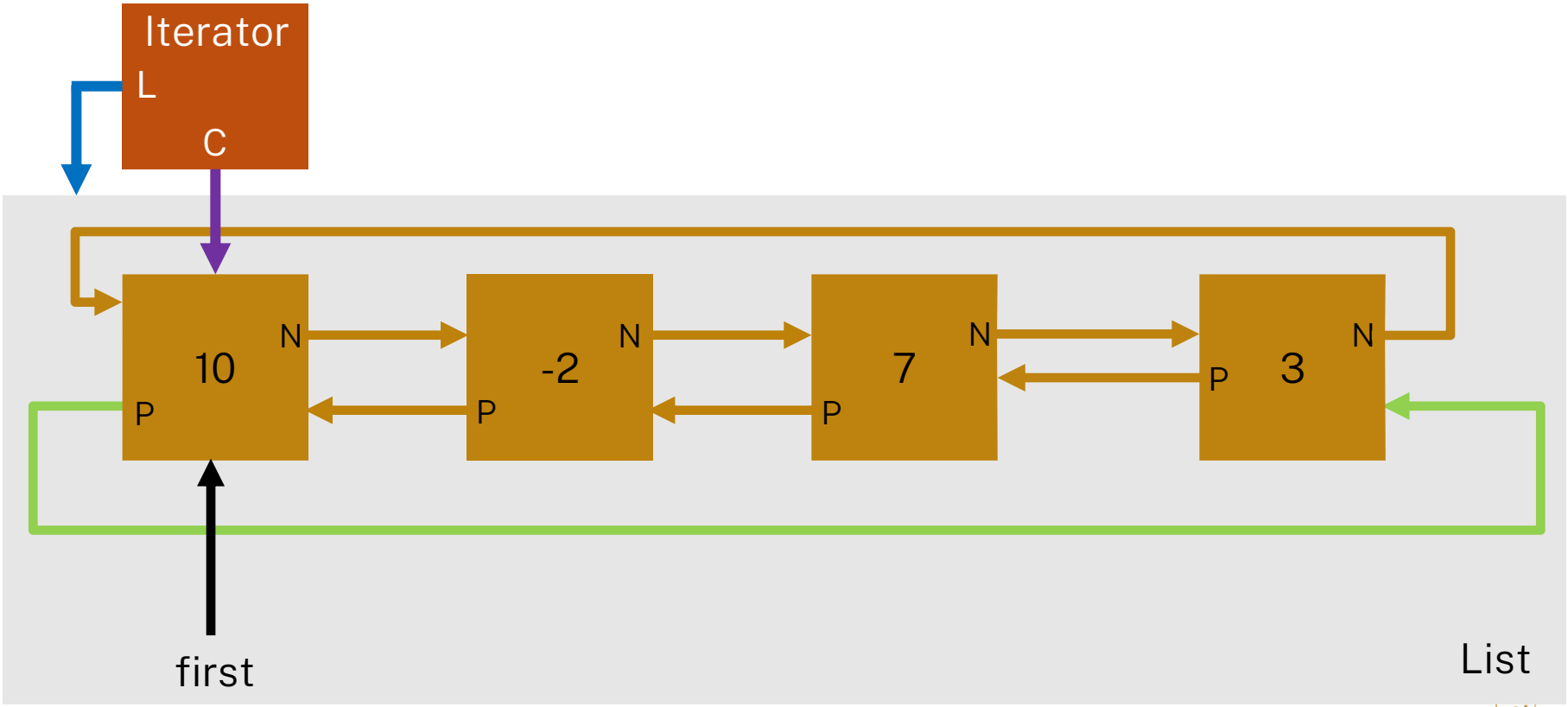
Inefficient: has to start from “first” every time and follow the “next” links i times.

Ideally: remember last list node, and carry on from there. But list nodes are private, for good reason.

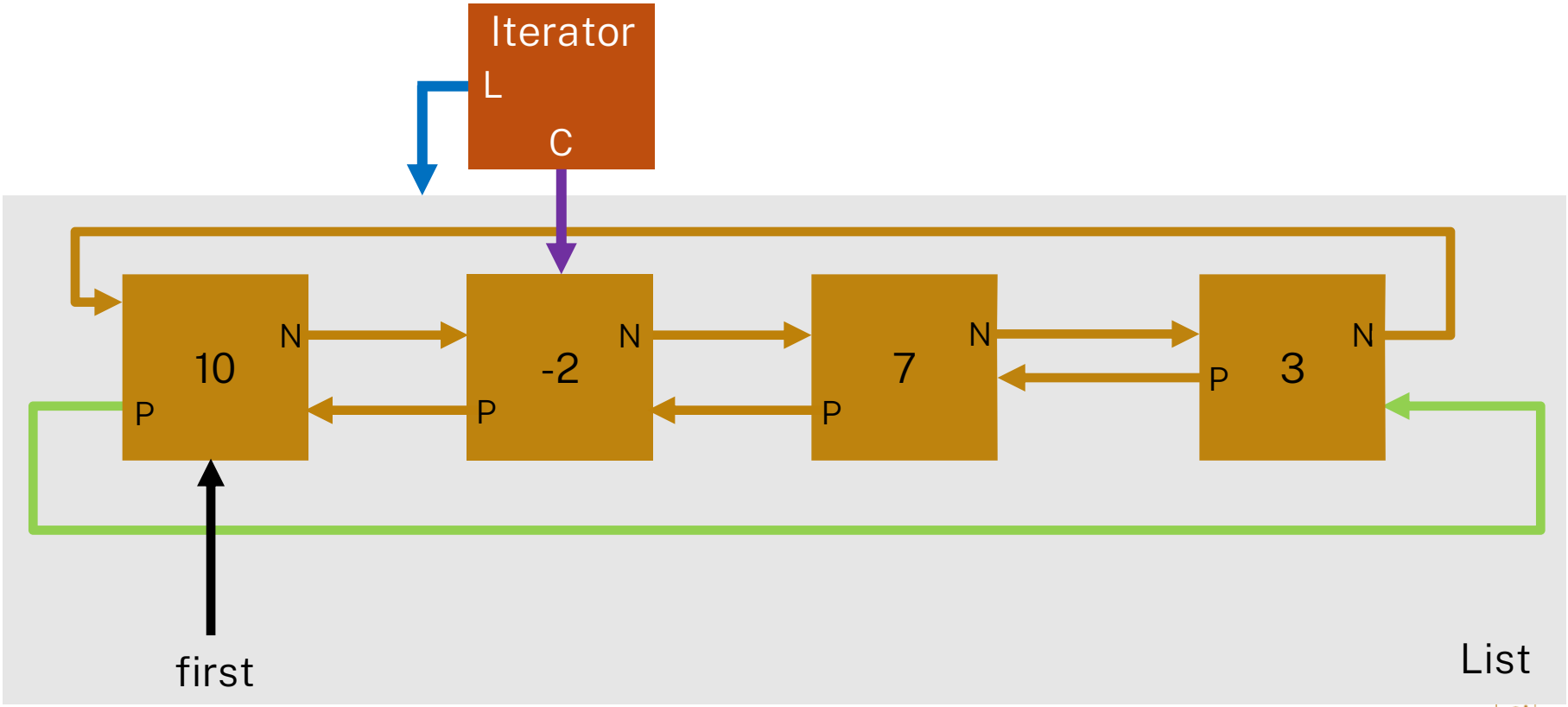
Doubly-Linked List Iterator



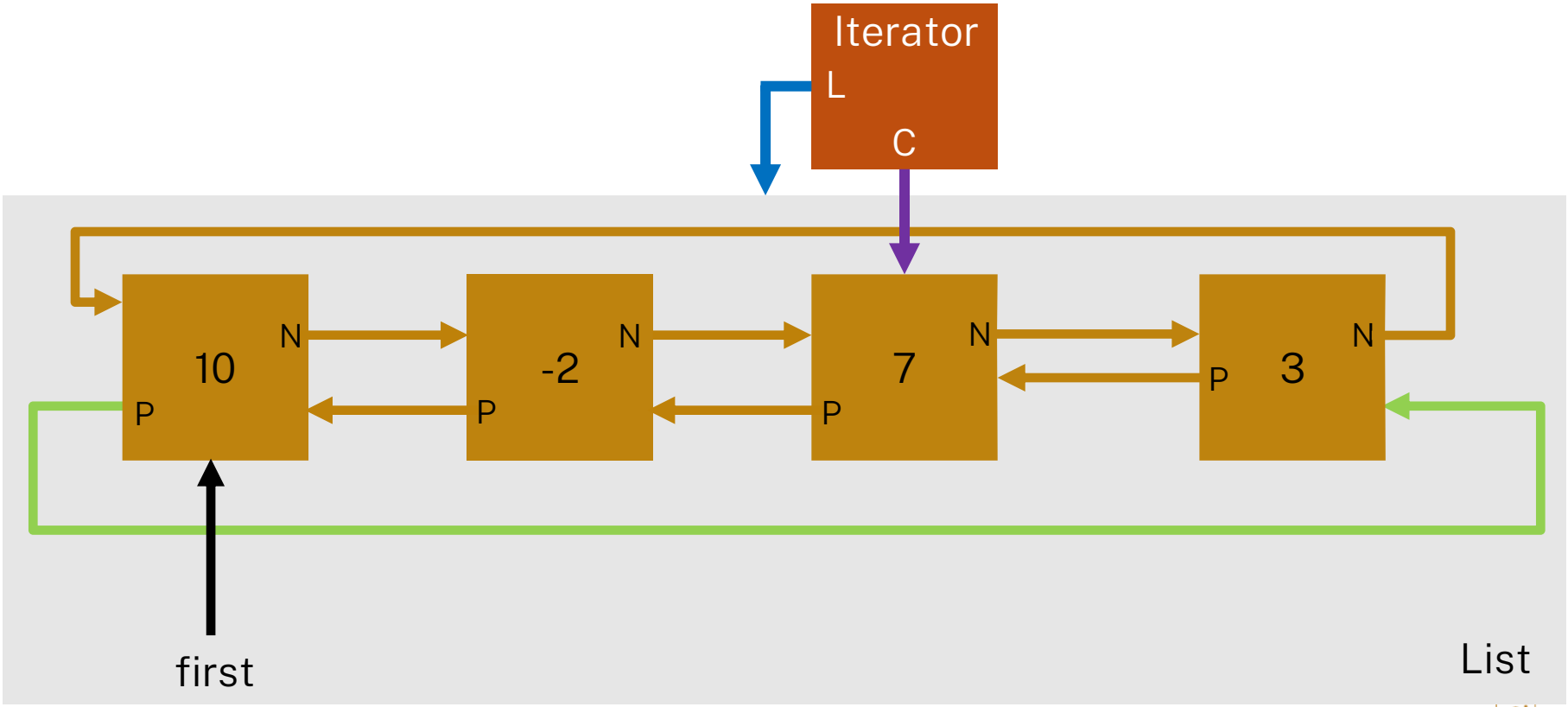
Doubly-Linked List Iterator



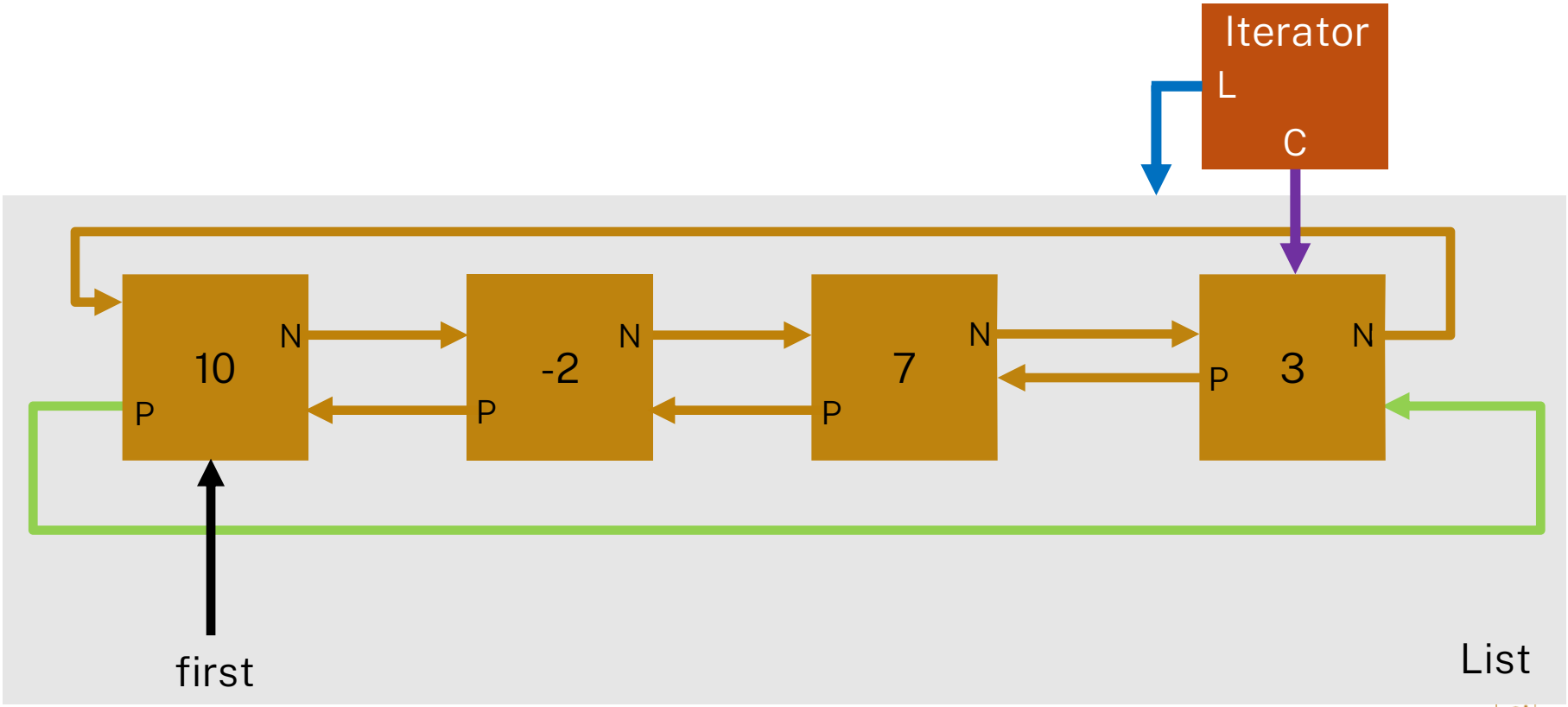
Doubly-Linked List Iterator



Doubly-Linked List Iterator



Doubly-Linked List Iterator



Iterable → Enhanced for-loops

```
for(var e : x) {
    …
}
```

This works whenever `x` has a type that implements `Iterable`

➔ Users don't need to know how to best iterate over x

Method Dispatch & Overloading

Australian National University

Overloading

Multiple methods with the same name

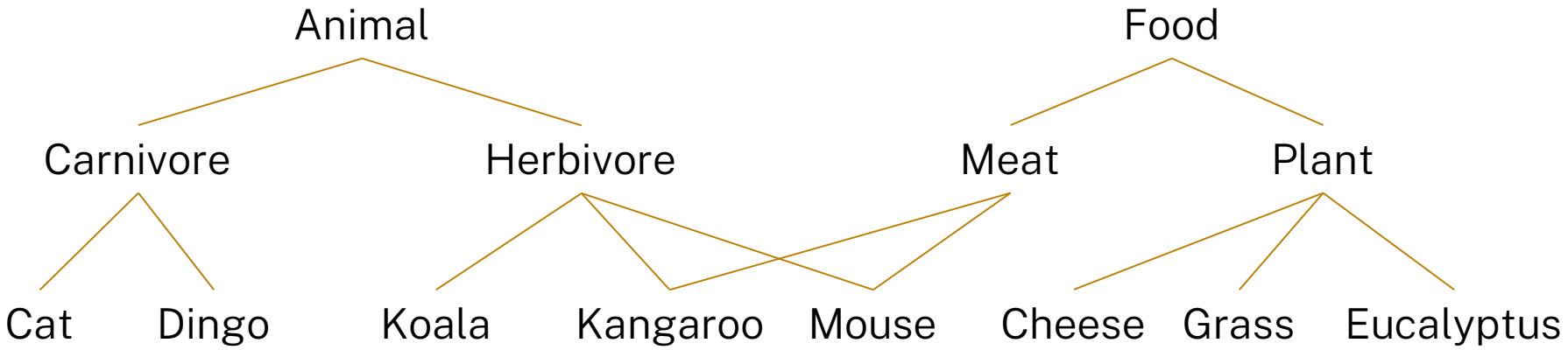
```
class Foo {
  void bar(int i) {
    System.out.println(i+1);
  }
  void bar(String s) {
    System.out.println(s);
  }
}
```

Distinguished by argument types

```
class Foo {
  void bar(int i) {
    System.out.println(i+1);
  }
  void bar(int i, int j) {
    System.out.println(i+j);
  }
}
```

Distinguished by argument count

“Animal-Kingdom-Oriented Programming”



Animal:	`eat(Food food)`
Carnivore:	`eat(Meat meat)`
Cat:	`eat(Mouse mouse)`

?

See ws8a/AnimalKingdomTests in IntelliJ demonstrations repo.

Recall: Multiple Phases

Java is a compiled language, meaning that creating programs has multiple steps. In Java:

Platform-Independent

Specialized to your Hardware/Operating System

➔ We distinguish between things that happen at “compile time” vs. things that happen at “run time”

*Terminology notes:
- “runtime”: an execution environment for your program, e.g. the Java Virtual Machine (the program you run as “java”)
- “run time”: the time when your program is executing
- “run-time X”: a thing that exists at run time
- “running time”: the time it takes for your program to run

Compile Time vs. Run Time

Compile Time

Things here are also referred to as “static” (related, but not the same as the `static` keyword in Java)

- “static type-checking”: finding type errors in your program without running it. This is one reason for why your program does not compile

Run Time

Things here are also referred to as “dynamic”, as opposed to “static”

- “dynamic type-checking”: finding type errors as your program is running (mostly in languages like Python and JavaScript, but some instances in Java). This is one reason for why your program crashes.

Compile Time vs. Run Time

```
Object o = “Hello World!”;
```

o’s compile-time type is `Object`

o’s run-time type is `String`

Compile Time vs. Run Time

Compile Time

Things here are also referred to as “static” (related, but not the same as the `static` keyword in Java)

- “static method dispatch”: for `static` (in the meaning we saw previously) method and constructor calls, we know at compile time which code is going to run

Run Time

Things here are also referred to as “dynamic”, as opposed to “static”

- “dynamic method dispatch”: for instance method calls, the exact code that is going to run depends on run-time type information (remember: *each object knows itself*).

Dynamic Method Dispatch

Single Dispatch

Most common form - used in Java, C++, C#, Python, JavaScript, etc:

Which code is executed only depends on **the run-time type of the receiver** and (where applicable) the compile-time types of the arguments.

Note: Python and JavaScript don't have overloading.

Multiple Dispatch

Rare form, used in e.g. Julia:

Which code is executed depends on both the run-time type of the receiver and the run-time types of the arguments.

Why rare? Hard to do efficiently, and hard to choose a "best" overloading.

Distinction-Level Content

Overloading vs Overriding

overriding = implementing/replacing a method in a subtype

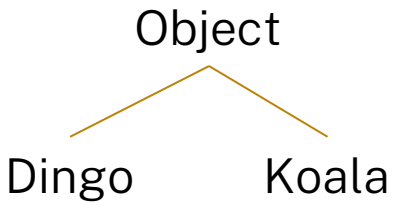
overloading = creating a truly different method with the same name

➔ Different overloadings get overridden separately

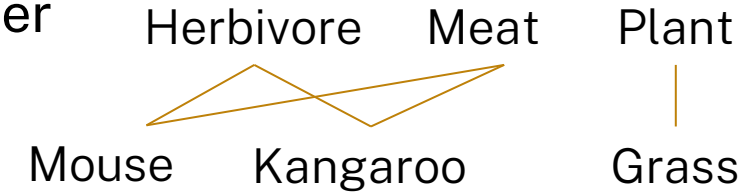
For each method call, overloading gets chosen at compile time based on static receiver and argument types. At run time, the overriding for that overloading is chosen based on the dynamic type of the receiver.

Structure of Inheritance/Subtyping

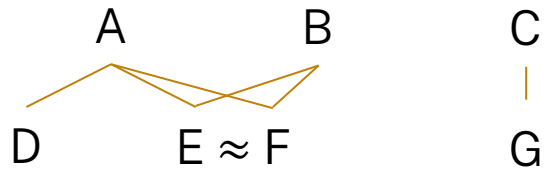
- Class inheritance forms a tree



- Interface inheritance forms a partial order
  A partial order is a reflexive, transitive, and antisymmetric relation



- Subtyping (in general) forms a preorder*
  A preorder is a reflexive and transitive relation



*the symmetries allowed in preorders are present in Java, but not covered in this course

Moving Up Loses Information

Recall: subtyping can be used to hide stuff

That cuts both ways!

```
Object o = “Hello World”;
```

`o` is a `String`, but you can’t use `o.length()`, because those members are now hidden.

Recovering Lost Type Information

Casts and instanceof

Australian National University

(Down-)Casts

```
Object o = …; //we somehow know that this is a String
String s = (String)o;
System.out.println(s.length());
```

A cast expression inserts a run-time check.

The check fails if o is not a String, which will crash the program. Conversely, if the program does not crash, o is a String, and so the assignment to s is valid.

instanceof

```
Object o = …;
if(o instanceof String) { … }
```

An instanceof expression checks whether the left operand has the type given as the right operand, and returns a corresponding boolean.

instanceof + casts

```
Object o = …;
if(o instanceof String) {
  String s = (String)o;
  …
}
```

```
Object o = …;
if(o instanceof String s) {
  …
}
```

The right-hand version is a (relatively new) shorthand for the common pattern on the left.

Generics

Avoiding Type Information Loss

Australian National University

Old Java List Interface

```
…
boolean add(Object o);
Object get(int index);
…
```

```
List l = new ArrayList();
l.add("Hello");
l.add(5);
String s = (String)l.get(0);
s = (String)l.get(1);
```

Need to write casts everywhere ☹
Need to hope that the values in the list are what you expect ☹
Easy to get wrong ☹

https://javaalmanac.io/jdk/1.4/api/java/util/List.html

Generics – Type Abstraction

Key Insight: for a typical container implementation (like lists), the type of its elements does not matter – it would do the same for any kind of element type. The user of the container, however, does care about what is in there.

Java's interpretation: the old implementations of ArrayList etc. do not need to change at all. Only the way they are type-checked.

Generic Type Arguments

- In interface/class/record declarations:
  list of type variables between angle brackets after name, e.g.:
  `public class Map<K,V> { … }`
  Scope: whole interface/class/record
- In static/instance method declarations:
  list of type variables between angle brackets before return type, e.g.:
  `public abstract <T> List<T> map(Function<String,T> fun);`
  Scope: method

Within scope, type argument is like any other type, except that we know nothing about its members (other than those from Object)

Generic Type Arguments

Main purpose: consistency

<T> T id(T t) { **return** t; } ✓☺ vs. Object id(Object t) { **return** t; } ✓☺

<T> T id(T t) { **return** 5; } ✗☺ vs. Object id(Object t) { **return** 5; } ✓☹

So user knows that id("Hello") will return a String, not some arbitrary thing.

WARNING – Type Erasure

In Java, generics is a compile-time concept only. The compiler turns

```
List<Integer> l = new ArrayList<>();
l.add(5);
Integer i = l.get(0);
```

into

```
List l = new ArrayList();
l.add(5);
Integer i = (Integer)l.get(0);
```

Just like in old Java, but with more guarantees, and automatically inserted casts.
➔ backwards-compatibility

You can also write this code directly, accessing the so-called “raw” types of List and ArrayList. Don’t do that!

WARNING – Type Erasure

Since the runtime does not know about generics, you cannot meaningfully write (where T is a type argument):

- `x instanceof T`
- `(T)x`

and

- `x instanceof List<T>` becomes `x instanceof List`
- `(List<T>)x` becomes `(List)x`

This is because at run time, we do not know anymore what T was

Generics and Subtyping

`String` is a subtype of `Object`.

Is `List<String>` a subtype of `List<Object>`?

NO!

```
List<Object> l = new ArrayList<Object>();
l.add(5); // this is fine
l = new ArrayList<String>(); // subtyping would allow this
l.add(5); // this would add an int to a list of Strings ☹
```


Advanced Generics & Subtyping

Distinction-Level Content

Subtyping - Variance

Variance = the relationship of movements in the subtyping hierarchy

Prime example: function subtyping

Covariance:

```
String myFun(Object o)
```

Since every String is an Object, this can be used wherever an

```
Object myFun(Object o)
```

is expected. Return types are “covariant” (return type is a subtype → function type is a subtype).

Subtyping - Variance

Variance = the relationship of movements in the subtyping hierarchy

Prime example: function subtyping

Contravariance:

```
Object myFun(Object o)
```

Since every String is an Object, this can be used wherever an

```
Object myFun(String o)
```

is expected. Argument types are “contravariant” (argument type is a supertype → function type is a subtype).

Subtyping - Variance

Variance = the relationship of movements in the subtyping hierarchy

Previous example: generics

Invariance:

`List<String>`

is not a subtype of

`List<Object>`

or vice versa.

Note: `ArrayList<String>` is a subtype of `List<String>`

Subtyping - Variance

Variance = the relationship of movements in the subtyping hierarchy

Intuition:

- A type argument is covariant if it is only used to “read” values
- A type argument is contravariant if it only use to “write” values
- A type argument is invariant if it is both used for “reading” and “writing”

Subtyping – Variance & Generics

Java allows wildcards ? in generic types, with modifiers for variance:

e.g. `List<?> x; List<? extends String> y; List<? super T> z;`

This is called “use-site variance”, as variance is declared in types, not type definitions.

Other languages have “declaration-site” variance, where a particular generic type is directly declared to always have a certain variance.

Subtyping – Variance & Generics

Java allows wildcards ? in generic types, with modifiers for variance:

`List<? extends T>` - covariance. If A subtype of B, then
`List<? extends A>` subtype of `List<? extends B>`

Disables access to methods with T in argument

Subtyping – Variance & Generics

Java allows wildcards ? in generic types, with modifiers for variance:

`List<? super T>` - contravariance. If A subtype of B, then
`List<? super B>` subtype of `List<? super A>`

Disables access to methods with T in return type

Subtyping – Variance & Generics

Java allows wildcards ? in generic types, with modifiers for variance:

`List<?>` is a list of anything. Any `List<T>` is a subtype of `List<?>`

Disables access to methods with T in argument, and makes T Object in return types.

Generics – Constraints

By default, we know nothing about a type argument T. Only methods from Object are available.

We can add constraints to type argument declarations in interfaces/classes/records/methods:

```
public class AnimalList<T extends Animal> {
    …
    void feedAll(Food food) {
        for(T t : animals) { t.eat(food); }
    }
```

T can be Cat, or Koala, … - up to Animal, so we know that Animal methods are available