# ICS 240: Introduction to Data Structures

Module 6b

Generics

## Running time Analysis of Algorithms

- Reading:
  - Chapter 1:
    - Section 1.2: pages 16 to 26

  - Appendix G: pages 805 - 806

## Generics

- Reading:
  - Chapter 5:
    - Section 5.1 – 5.3

  - Appendix G

## Main points

- Review of wrapping / unwrapping primitives
- Narrowing and widening conversions
- Generic methods
  - “<E>” and “<T>”
- More on Iterators and Iterable interface

## Generic Programming– High Level Overview

- declare and use Java Object variables that can refer to any kind of object in a  Java program.
- correctly use widening and narrowing to convert back and forth between Java  Object variables and other object types.
- correctly use autoboxing and unboxing to convert back and forth between  Java's primitive types and the Java wrapper classes.
- design, implement, and use generic methods that depend on a generic type  parameter that varies from one activation to another.
- design, implement, and use generic collection classes where the data type of  the underlying elements is specified by a generic type parameter.
- create classes that implement interfaces and generic interfaces from the Java  API.
- design, implement, and use Java iterators for a collection class.
- use Java's enhanced forloop for collections with iterators.
- find and read the API documentation for the Java classes that implement the  Collection interface (such as ArrayList) and the Map interface (such as  TreeMap) and use these classes in your programs.

## Motivation

- There are a few types of primitives, but there are orders of magnitude more types of objects (including ones you can create yourself).
- It would be nice if one collection class (say, an ArrayList class) could be written that would work for all types of objects.
  - It would be a generic class, it works for any object.

## Widening Conversions

- Java has a built-in Object class.
  - Object is a superclass of all classes of objects.
  - An Object variable can hold a reference to any type of object.
    - E.g., a Clock, a Node, etc.

```
Node n = new Node( value, null );
Object objNode = n;     // Widening conversion
```

- So now we have two references to the same object.
  - One of type Node, one of type Object.
- Whenever we convert an object of a subclass (e.g., Node) to an object of its superclass (e.g., Object), this is called a *widening conversion.*
- Why:  One reason is that we may not want to pass this to another method, but we don’t want that other method to be able to mess around with subclass fields.

# Widening and Narrowing

```
Node n = new Node( value, null );
Object objNode = n;     // Widening conversion
```

$$\begin{tikzcd}\text{Node n} \arrow[r] & \boxed{\textit{Something}} & \text{Object objNode} \arrow[l]\end{tikzcd}$$

- At some later point, we may want to pass that Object around to some other method that wants to treat it as a Node again.
- This is called a *narrowing conversion*.
- We can only do this if the object is a Node, so we have to explicitly *cast* the object to the appropriate subtype.

```
Node q = (Node) objNode;   // Narrowing conversion
Object objNode = n;        // Widening conversion
```

- Some method return Object, and a narrowing cast may be necessary to use that object.
- If a narrowing conversion is invalid, you get a ClassCastException at runtime.

# Wrapper Classes

- Many methods are designed to work with objects.
- Ideally, you could write <u>one</u> method that would work with <u>any</u> type of object.
  - But then how can you get it to work with primitives?
- You "wrap" the primitive in an object meant for that particular primitive.
  - There is one wrapper class per primitive.

| **Primitive** | **byte** | **short** | **int** | **long** | **float** | **double** | **char** | **boolean** |
|---|---|---|---|---|---|---|---|---|
| Wrapper | Byte | Short | Integer | Long | Float | Double | Character | Boolean |

- Example: java.lang.Integer

# Boxing and Unboxing primitives

- Note: The text uses a deprecated boxing strategy.
- Primitive → Wrapper = Boxing
- Wrapper → Primitive = Unboxing

```
int j = 17;
Integer k = Integer.valueOf( j );   // Boxing
int m = k.intValue();  // Unboxing
```

- Boxing and unboxing can be automatic in some cases.
  - If Java expects an object of a wrapper class and it gets a primitive of the appropriate type, it autoboxes.
  - If Java expects a primitive and it gets a wrapped primitive of the appropriate type, it auto-unboxes.

```
int j = 17;
Integer k = j; // Autoboxing
int m = k;  // Autounboxing
```

# Generic methods

- Example (text): I want to find the middle element of an array of objects of any type. I do this by writing the method to take as a parameter a *generic* object.
  - A generic object is enclosed in angle brackets in the signature of a method, before the return type.
    - Traditionally, the generic object is referred to as "T"

```
public static <T> T middle( T[] data ) {
    if ( data.length == 0 ) {
        return null;
    }
    else {
        return data[ data.length / 2 ];
    }
}
```

## What’s that extra token in the signature line?

```
public static <T> T middle( T[] data ) {
     if ( data.length == 0 ) {
          return null;
     }
     else {
          return data[ data.length / 2 ];
     }
}
```

## What’s that extra token in the signature line?

```
  1        2      3   4    5       6       7
public static <T> T middle( T[] data ) {
    if ( data.length == 0 ) {
        return null;
    }
    else {
        return data[ data.length / 2 ];
    }
}
```

1. Visibility modifier
2. Static method (default: instance method)
3. Token stating that “T” is a generic type
4. Return type
5. Method name
6. Parameter type
7. Parameter name

## Generic methods at compilation & run time

- At compile time, the compiler can see what the type <T> is supposed to be in a particular call to the middle(…) method.
  - The data type must always be a class type, not a primitive type
    - That’s one thing wrapper classes are good for
- Note that the exact data type of a generic type is unknown at run time. This is called “erasure”.

# Writing a Generic Method

- Textbook example, return the middle element of an array:

static <T> T middle( T[] data ) …

- If you pass it an array of one type of object and then you call it in the same method with a return of something else, the compiler will complain.

```
Integer[] a1;
Double middleDouble;

middleDouble = middle( a1 );
```

# Generic Classes

- Here is the power. We can write one class that can handle objects of any type.
  - Including wrapped primitives.
- Especially useful for collection classes.
- So, for instance, we can write an ArrayBag<E> class that stores elements of type E in the bag.

```
public class ArrayBag<E> implements Cloneable
{
   // Invariant of the ArrayBag class:
   //   1. The number of elements in the bag is in the instance
   //      variable manyItems, which is no more than data.length.
   //   2. For an empty bag, we do not care what is stored in any of
   //      data; for a non-empty bag, the elements in the bag are
   //      stored in data[0] through data[manyItems-1], and we don’t
   //      care what's in the rest of data.
   private Object[ ] data;
   private int manyItems;
```

- We have to declare the data as objects because a program cannot create an array where the components are a generic type parameter.

## Methods of a Generic Class

- They look the same as they did with a primitive, and the same way they would with an object.
- Only you use the generic name as the parameter type.
  - The code for the following methods (and some others) is unchanged from IntArrayBag, only the signatures are changed.

```
public void add( E element ) { …… }
public void addAll( ArrayBag<E> addend ) {…}
public void addMany( E… elements ) {…}
public int countOccurrences( E target ) {…}
```

## Declaring Instances of a Generic Class

```
ArrayBag<String> stringBag = new ArrayBag<String>();
ArrayBag<Double> doubleBag = new ArrayBag<Double>();
```

## Returning Objects from the Collection

- When we return an Object from the collection of Objects, we need to cast it to the generic type.

```
public E grab( )
  {
     int i;

     if (manyItems == 0)
        throw new IllegalStateException("Bag size is zero");

     i = (int)(Math.random( ) * manyItems);
     return (E) data[i];
  }
```

## Comment on warnings

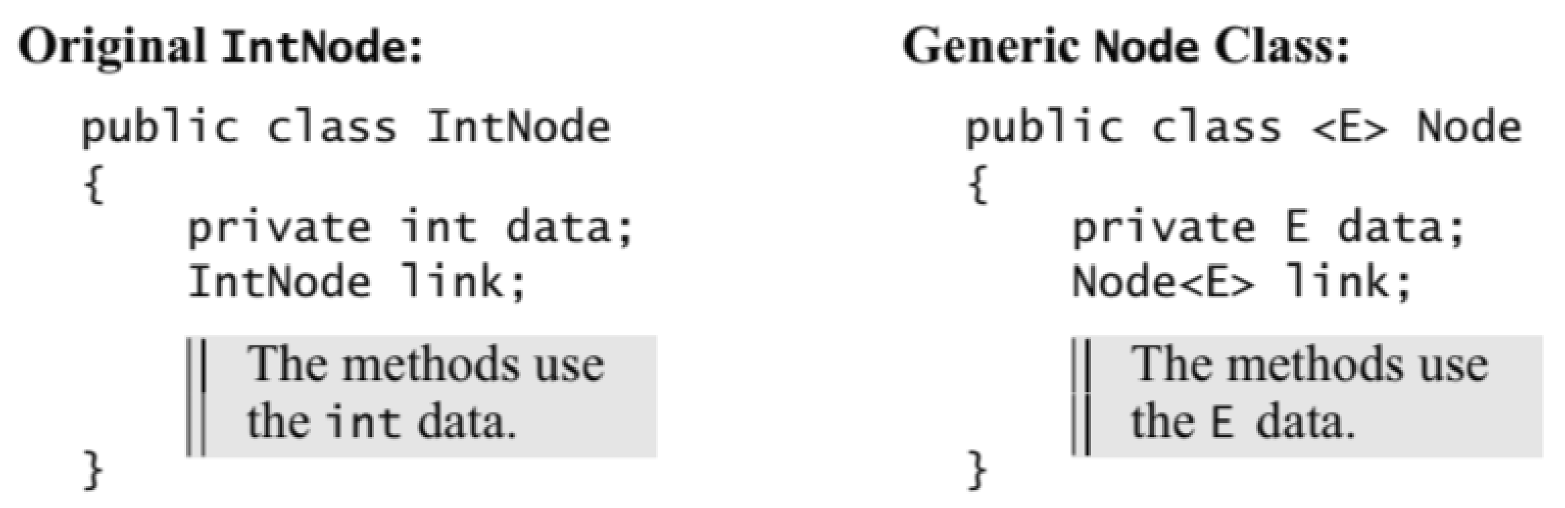
- The text states that when you compile the ArrayBag class you will get compiler warnings.
- To avoid that, the class has a compiler directive
  - @suppressWarnings(“unchecked”)
  - This warning alerts you that the compiler an’t determine that the data being passed are all of type E.
  - So it’s OK to suppress the warnings as they do in the code, but be careful that you only pass elements of the correct type.

# Summary:  Making Collection Class Generic

1. Change the name of the class to contain the generic.
2. Change the type of the data array to hold Objects
3. Change the types of the underlying elements to generic elements.
4. Change Static Methods to Generic Static Methods
5. Typecast when an element is retrieved from the collection
6. Suppress warnings
7. Update equality tests to check for object equality
8. Decide how to handle null references
9. Set unused reference variables to null
10. Update documentation

## Generic Nodes

- As noted succinctly in the text, just change the type of data from primitive to generic type.
- And use the type E rather than the primitive type in operations.



**Original IntNode:**

```
public class IntNode
{
    private int data;
    IntNode link;

    The methods use
    the int data.
}
```

**Generic Node Class:**

```
public class <E> Node
{
    private E data;
    Node<E> link;

    The methods use
    the E data.
}
```

## Iterator interface

- An iterator is a way to step through all the elements of the class. Iterators are usually implemented using generics.

```
public interface Iterator<E> {
      public Boolean hasNext();
      public E next();
      public void remove();  // optional
```

- The remove method doesn’t always make sense.
  - For instance, on a read-only collection.
- Every class in the Java Collections Framework implements Iterator.
  - They provide an iterator() ethod that returns an instance of an Iterator over the elements in that collection.

## Iterator vs Iterable

- There is also a separate Iterable interface that allows an object to be the target of an enhanced for statement (the “for-each” loop).
- This is something different, we don’t talk about it here.
  - It’s in section 5.6 of the textbook if you’re interested.

# Subclasses of Iterator

- You can see that there are subclasses of Iterator.
- ListIterator allows you to iterate through a List forwards or backwards. I won’t explicitly talk about it, but you can try it if you wish.
- There are also Iterators for primitives.
- Lots of stuff you can do if you want to look up the API and play with it.
  - All are OK to use in this course.
  - None are required.

## How to Traverse a List

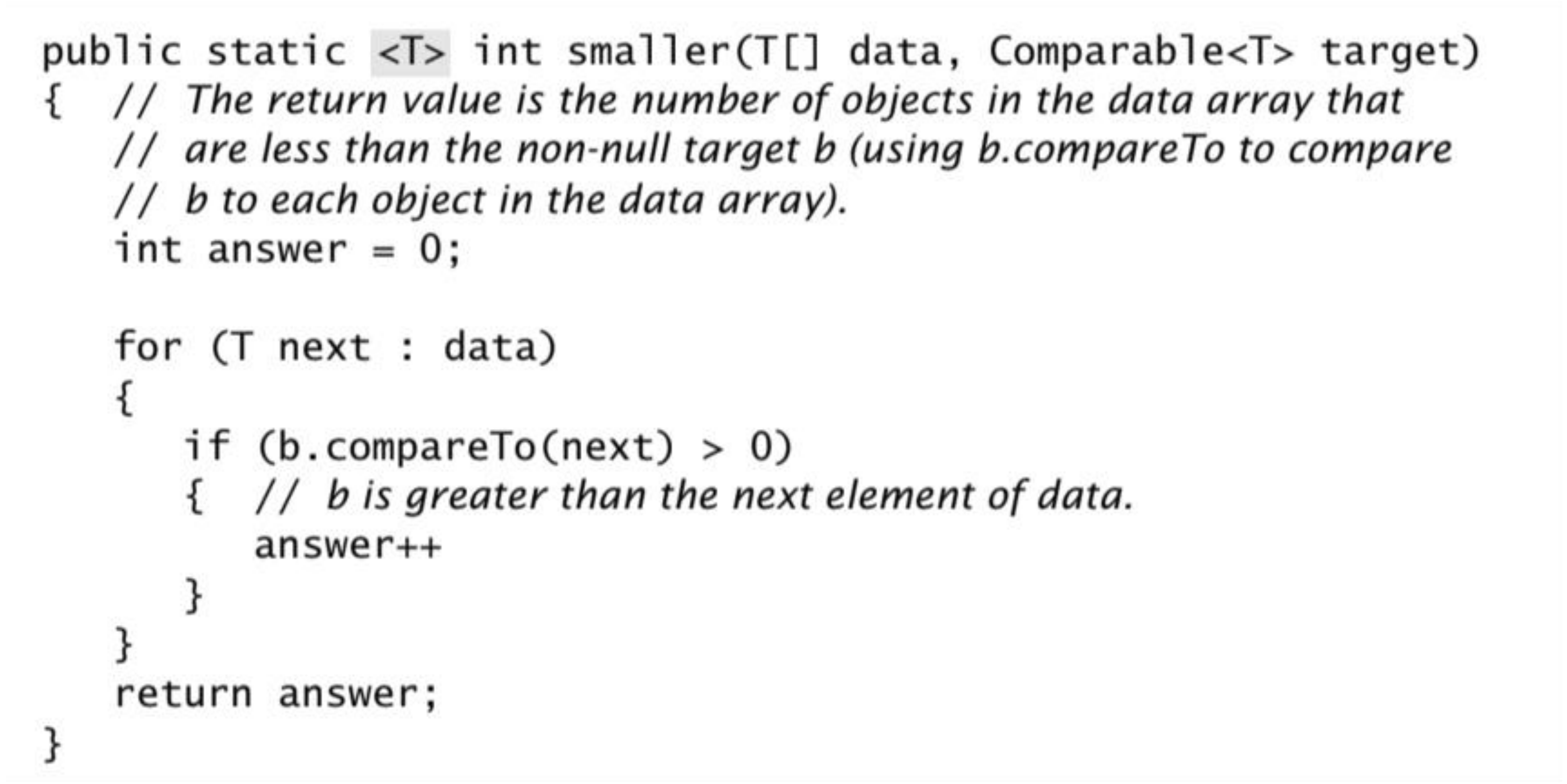
- List is an interface that is implemented by things like
  - ArrayList<E>
  - LinkedList<E>
- One of my favorite internet links is crunchify's illustration of the many ways to iterate through a list.

## Why Use Iterators

- Big gain: They make it easier to remove elements while iterating through a list.
- As you found out in Program B, when you're removing elements from a Sequence (a type of List, basically), you have to be very careful not to try to read beyond the ever-changing last index of the sequence.
- Iterators handle this problem for you.
- Using an Iterator can also be sometimes faster, but not slower, than "growing your own" loop to iterate over a collection.

# Using an Interface as Type of a Parameter

- When an interface name is used as the type of a method's parameter, the actual argument must be a data type that implements the interface.



```
public static <T> int smaller(T[] data, Comparable<T> target)
{  // The return value is the number of objects in the data array that
   // are less than the non-null target b (using b.compareTo to compare
   // b to each object in the data array).
   int answer = 0;

   for (T next : data)
   {
      if (b.compareTo(next) > 0)
      {  // b is greater than the next element of data.
         answer++
      }
   }
   return answer;
}
```

# How do you know if a class implements an interface?

- You can find out whether or not a given object is of a class that implements a given interface by using the instanceof method
  - A boolean method that returns true if the class implements the interface, and false if it does not.
- You can also use it to test if an object is an instance of a given class

String s = “Hello.”;

….<far, far away in another method>

```
if ( s instanceof String )
    System.out.println( s );
else
      // something else
```