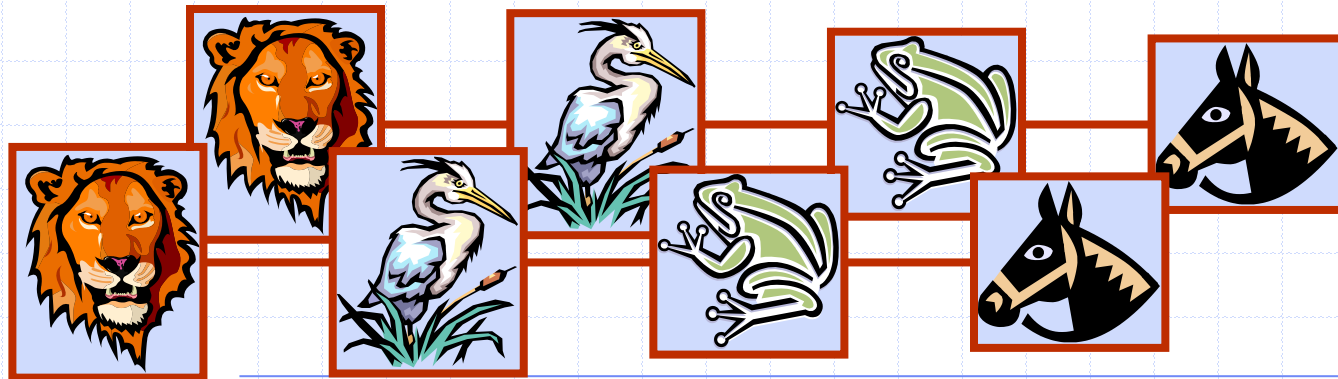


Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Equivalence Testing and Cloning Data Structures



Equivalence Testing

At the Lowest Level

- When working with reference types, there are many different notions of what it means for one expression to be equal to another.
- At the lowest level, if a and b are reference variables, then expression
 - $a == b$
 - tests whether a and b refer to the same object (or if both are set to the null value).

At a Higher Level

- However, for many types there is a higher-level notion of two variables being considered “equivalent” even if they do not actually refer to the same instance of the class.
- For example, we typically want to consider two String instances to be equivalent to each other if they represent the identical sequence of characters.

The Object Class Method equals()

- To support a broader notion of equivalence, all object types support a method named equals.
- Users of reference types should rely on the syntax `a.equals(b)`, unless they have a specific need to test the more narrow notion of identity.
- The equals method is formally defined in the Object class.
- But that implementation reverts to returning the value of expression
 - `a == b`.
- Defining a more meaningful notion of equivalence requires knowledge about a class and its representation.

Implementing equals() Method

- The author of each class has a responsibility to provide an implementation of the equals method, which overrides the one inherited from Object, if there is a more relevant definition for the equivalence of two instances.
- For example, Java's String class redefines equals to test character-for-character equivalence.

Equivalence Relation

- When overriding the notion of equality, equivalence relation should satisfy the following properties:
- **Treatment of null:** For any nonnull reference variable x , the call $x.equals(\text{null})$ should return false (that is, nothing equals null except null).
- **Reflexivity:** For any nonnull reference variable x , the call $x.equals(x)$ should return true (that is, an object should equal itself).
- **Symmetry:** For any nonnull reference variables x and y , the calls $x.equals(y)$ and $y.equals(x)$ should return the same value.
- **Transitivity:** For any nonnull reference variables x , y , and z , if both calls $x.equals(y)$ and $y.equals(z)$ return true, then call $x.equals(z)$ must return true as well.

Equivalence Testing with Arrays

- The `java.util.Arrays` class provides a summary of the treatment of equivalence for arrays, assuming that variables `a` and `b` refer to array objects:
- `a == b`: Tests if `a` and `b` refer to the same underlying array instance.
- `a.equals(b)`: This is identical to `a == b`. Arrays are not a true class type and do not override the `Object.equals` method.
- `Arrays.equals(a,b)`: This provides a more intuitive notion of equivalence, returning true if the arrays have the same length and all pairs of corresponding elements are “equal” to each other.
- More specifically, if the array elements are primitives, then it uses the standard `==` to compare values. If elements of the arrays are a reference type, then it makes pairwise comparisons `a[k].equals(b[k])` in evaluating the equivalence.
- For most applications, the `Arrays.equals` behavior captures the appropriate notion of equivalence.

Equivalence Testing with Linked Lists

- Two lists are equivalent if they have the same length and contents that are element-by-element equivalent.
- We can evaluate such equivalence by simultaneously traversing two lists, verifying that: `x.equals(y)`
- for each pair of corresponding elements `x` and `y`.
- The implementation of the `SinglyLinkedList.equals` method is given in Code Fragment 3.19.
- In Code Fragment 3.19, `getClass( )` method is supported by all objects to test whether the two instances belong to the same class

```

1  public boolean equals(Object o) {
2      if (o == null) return false;
3      if (getClass() != o.getClass()) return false;
4      SinglyLinkedList other = (SinglyLinkedList) o;      // use nonparameterized type
5      if (size != other.size) return false;
6      Node walkA = head;                                  // traverse the primary list
7      Node walkB = other.head;                             // traverse the secondary list
8      while (walkA != null) {
9          if (!walkA.getElement().equals(walkB.getElement())) return false; //mismatch
10         walkA = walkA.getNext();
11         walkB = walkB.getNext();
12     }
13     return true;    // if we reach this, everything matched successfully
14 }

```

Code Fragment 3.19: Implementation of the SinglyLinkedList.equals method.

Cloning Data Structures

Cloning Data Structures

- In a programming environment, a common expectation is that a copy of an object has its own state and that, once made, the copy is independent of the original.
- This implies that changes to one do not directly affect the other.
- However, when objects have fields that are reference variables pointing to auxiliary objects, it is not always obvious whether a copy should have a corresponding field that refers to the same auxiliary object, or to a new copy of that auxiliary object.

The Object Class clone() Method

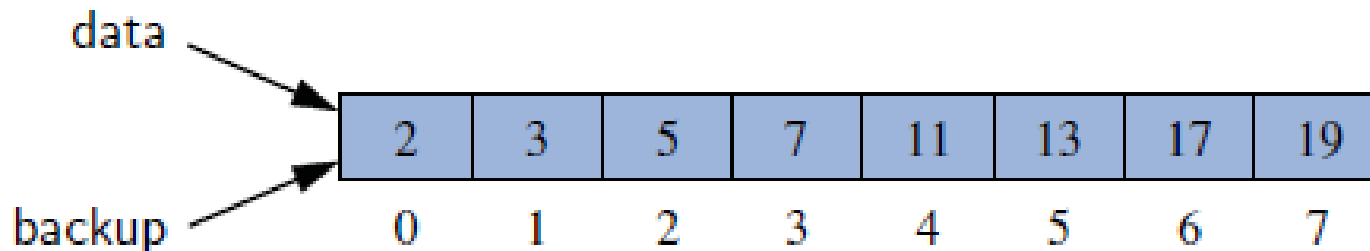
- Each class in Java is responsible for defining whether its instances can be copied, and if so, precisely how the copy is constructed.
- The universal Object superclass defines a method named clone, which can be used to produce what is known as a shallow copy of an object.
- Clone method: This uses the standard assignment semantics to assign the value of each field of the new object equal to the corresponding field of the existing object that is being copied.
- A shallow copy of an object: If the field is a reference type, then an initialization of the form
`duplicate.field = original.field`
causes the field of the new object to refer to the same underlying instance as the field of the original object.

Cloning Data Structures

- Shallow copy is not always appropriate for all classes, and therefore, Java intentionally disables use of the `clone( )` method by declaring it as protected, and by having it throw a `CloneNotSupportedException` when called.
- The author of a class must explicitly declare support for cloning by formally declaring that the class implements the `Cloneable` interface, and by declaring a public version of the `clone( )` method.
- That public method can simply call the protected one to do the field-by-field assignment that results in a shallow copy, if appropriate.
- However, for many classes, the class may choose to implement a deeper version of cloning, in which some of the referenced objects are themselves cloned.

Cloning Arrays that Store Primitive Types

- As a first example, consider the following code:
- `int[ ] data = {2, 3, 5, 7, 11, 13, 17, 19};`
- `int[ ] backup;`
- `backup = data; // warning; not a copy`
- The assignment of variable `backup` to `data` does not create any new array; it simply creates a new alias for the same array, as portrayed in Figure 3.23.



© 2014 C **Figure 3.23:** The result of the command `backup = data` for `int` arrays.

Cloning Arrays that Store Primitive Types

- Instead, if we want to make a copy of the array, data, and assign a reference to the new array to variable, backup, we should write:
- `backup = data.clone( )`
- The clone method, when executed on an array, initializes each cell of the new array to the value that is stored in the corresponding cell of the original array. This results in an independent array, as shown in Figure 3.24.

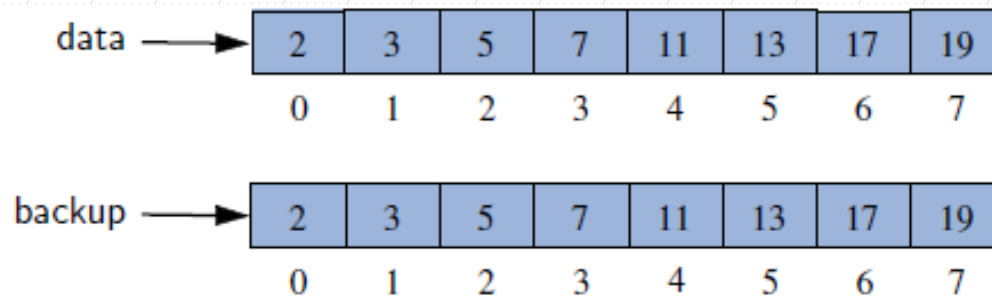


Figure 3.24: The result of the command `backup = data.clone()` for `int` arrays.

Cloning Arrays that Store Reference Types

- There are more considerations when copying an array that stores reference types rather than primitive types.
- The `clone()` method produces a shallow copy of the array, producing a new array whose cells refer to the same objects referenced by the first array.
- For example, if the variable `contacts` refers to an array of hypothetical `Person` instances, the result of the command
`guests = contacts.clone()`
- produces a shallow copy, as portrayed in Figure 3.25.

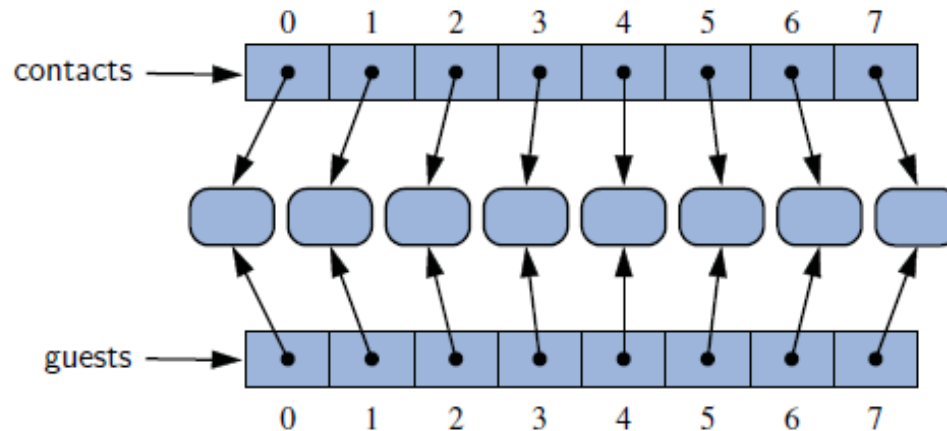


Figure 3.25: A shallow copy of an array of objects, resulting from the command `guests = contacts.clone()`.

Cloning Arrays that Store Reference Types

- A deep copy of the contact list can be created by iteratively cloning the individual elements, as follows, but only if the Person class is declared as Cloneable.
- `Person[ ] guests = new Person[contacts.length];`
- `for (int k=0; k < contacts.length; k++)`
- `guests[k] = (Person) contacts[k].clone( );`
- `// returns Object type`

Cloning Linked Lists

- The first step to making a class cloneable in Java is declaring that it implements the Cloneable interface.
- Therefore, we adjust the first line of the class definition to appear as follows:

```
public class SinglyLinkedList<E> implements Cloneable {
```
- The remaining task is implementing a public version of the clone() method of the class, which we present in Code Fragment 3.21.
- By convention, that method should begin by creating a new instance using a call to super.clone(), which in our case invokes the method from the Object class (line 3).
- Because the inherited version returns an Object, we perform a narrowing cast to type SinglyLinkedList<E>.
- At this point in the execution, the other list has been created as a shallow copy of the original.

Cloning Linked Lists

- Since our list class has two fields, size and head, the following assignments have been made:
other.size = this.size;
other.head = this.head;
- While the assignment of the size variable is correct, we cannot allow the new list to share the same head value (unless it is null).
- For a nonempty list to have an independent state, it must have an entirely new chain of nodes, each storing a reference to the corresponding element from the original list.
- We therefore create a new head node at line 5 of the code, and then perform a walk through the remainder of the original list (lines 8–13) while creating and linking new nodes for the new list.

```

1  public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
2      // always use inherited Object.clone() to create the initial copy
3      SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // safe cast
4      if (size > 0) { // we need independent chain of nodes
5          other.head = new Node<>(head.getElement(), null);
6          Node<E> walk = head.getNext(); // walk through remainder of original list
7          Node<E> otherTail = other.head; // remember most recently created node
8          while (walk != null) { // make a new node storing same element
9              Node<E> newest = new Node<>(walk.getElement(), null);
10             otherTail.setNext(newest); // link previous node to this one
11             otherTail = newest;
12             walk = walk.getNext();
13         }
14     }
15     return other;
16 }

```

Code Fragment 3.21: Implementation of the SinglyLinkedList.clone method.