Generic Programming

*Really* reusable code
First, a bit of history ...

• Since Java version 5.0, Java has borrowed a page from C++ and offers a template mechanism, allowing programmers to create data structures without specifying an underlying data type

• Until version 5.0, the closest Java had to this mechanism were the Object data type and wrapper classes

• We’ll examine these older (but still viable) solutions first
Object class

• Mother of all Java classes
• Except for the primitive types, all data types inherit from Object
• Consequently, an Object reference (variable) can refer to any Java object
Good Example

Object ob1 = new Random();
   // this is a **widening** type conversion – the data
   // type of the reference is broader than that of
   // the object it refers to
Random rg = (Random)ob1;
   // This explicit type cast is a **narrowing** conversion
Bad Example

Object ob2 = new String ("uh-oh");
   // So far, so good …
rg = (Random)ob2;
   // D’oh! ClassCastException is thrown – since the
   // underlying object is a String, it can’t be arbitrarily
   // cast as a completely unrelated data type
Always a bad example?

• The previous example (assignment of object to Object, subsequent cast of Object reference to another type) doesn’t work

• Are there circumstances under which such a combination would work?
The clone() method

• Inherited by all data types from Object
• Can be (and often is) overridden, as in the example below:

```java
public IntNode clone() {
    IntNode n;
    n = (IntNode)super.clone();
    ...
}
```
Wrapper classes

• Wrapper classes are Java’s workaround mechanism for the fact that primitive data type variables aren’t objects

• They allow easy conversion from primitive data to object data and vice-versa

• Each primitive type has corresponding wrapper – e.g. Integer for int, Double for double, etc.
Familiar example

Scanner kb = new Scanner(System.in);
String input;
int num;
System.out.print(“Enter a number: ”);
input = kb.nextLine();
um = Integer.parseInt(input);
Data conversion with wrappers: boxing and unboxing

```java
int n1 = 29;
Integer nWrapper = new Integer(n1);
    // n1 is “boxed” inside the wrapper
int n2 = nWrapper.intValue();
    // value is “unboxed” and assigned to primitive
    // variable
```
Boxing and unboxing syntax is simplified in Java 5.0

```java
int n1 = 29;
Integer nWrapper = n1;
int n2 = nWrapper;
```

- Although the syntax is simplified, the same operations (from previous slide) still have to be performed – and take just as much time
- Prior to Java 5.0, this code would produce compiler errors
Example: pre-Java 5.0 generic method

```java
public static Object findMax (Object [] data) {
    Object max = data[0];
    for(int x=1; x<data.length; x++)
        if (data[x].compareTo(max) > 0)
            max = data[x];
    return max;
}
```
Disadvantages of previous code

• The “compareTo” method must be meaningful for the underlying data type
• More important: mishandling of the return value from this method could throw a ClassCastException
The modern way: generic methods

• Parameter type(s) not fully specified
• Uses <> syntax to indicate generic method; in this, as in other aspects, the mechanism resembles the same one in C++
• In C++, the mechanism is called templates – hence the use of the letter ‘T’ in subsequent examples
Generic method example

```java
public static <T> T findMax(T[] data) {
    T max = data[0];
    for (int x=1; x<data.length; x++)
        if (max.compareTo(data[x]) < 0)
            max = data[x];
    return max;
}

// Note: we assume this method is a static member
// of a class named ArrayClass
```
Syntax notes

• We want the method to be able to return (or take as an argument) any data type
• The <T> appearing before the return type in the heading indicates that the use of T as a data type (both as return type and in parameter list) refers to some specific data type, but we don’t care which one
Some restrictions apply ...

– T must be a descendant from Object, not a primitive type
– T (for this example method) must implement the Comparable interface
– You can’t call a constructor for a generic type
– You can’t create a new array of a generic type
Finding max using original (Object) method

Random rg = new Random();
int [] nums = new int [100];
Integer [] onums = new Integer[100];
for (int x=0; x<nums.length; x++) {
    nums[x] = rg.nextInt();
    onums[x] = new Integer(nums[x]);
}
int largest =
((Integer)ArrayClass.findMax(onums)).intValue();
    // type cast required – could throw ClassCastException
Finding max using improved (generic) method

Random rg = new Random();
int [] nums = new int [100];
Integer [] onums = new Integer[100];
for (int x=0; x<nums.length; x++) {
    nums[x] = rg.nextInt();
    onums[x] = new Integer(nums[x]);
}
int largest = ArrayClass.findMax(onums);
    // no type cast required – type checking takes place at compile time, meaning no runtime error
Generic classes

• Extend generic idea to entire class, not just single method
• Underlying data type in collection class (such as Bag or Sequence) can be left unspecified until the class is instantiated
Syntax for generic classes

• Essentially same as for method; example:
  
  ```java
  public class Bag<T> implements Cloneable {
      private T[] data;
      int manyItems;
      int defaultSize = 10;
      ...
  }
  ```

• When class is instantiated, replace generic name T with real data type:
  
  ```java
  Bag<String> b = new Bag<String>();
  ```
Generic classes and arrays

• Problem: Can’t create an array without a known data type

• Old solution:
  • Can get around this by creating an array of Objects, then casting, as shown below:
    public Bag() {
      manyItems = 0;
      data = (T[]){new Object[defaultSize];
    }
  • This code produces a compiler warning, but still may compile (then again, may not); problem is, compiler doesn’t know what T is
Generic classes & arrays

• New (& better) solution:
  – Declare “generic” array as an array of Objects
  – Perform the typecast upon retrieval of individual data item, not on entire array
  – Still may produce compiler warnings, but is much more reliable – example (from generic ArrayBag):

    ```java
    while ((index < manyItems) && (!target.equals((E)data[index])))
    index++;
    ```
Generic class type as method parameter

• In the Bag class, we had a method called addAll, which took a Bag argument and added the argument’s contents to the calling Bag object:
  public void addAll (Bag addend)

• We can create a generic version of this method (assuming it is part of the generic class), as follows:
  public void addAll (Bag <T> addend)
Cautionary notes on generic collection classes

• You have a collection of references, not a collection of data

• It’s good practice in such a situation to set any unused references to null – because this is an empty “value” for which we can test, in order to avoid runtime errors
Converting a collection class to a generic collection class

1. Change its name:
   public class Name becomes public class Name<T>

2. Change underlying data type of instance arrays to Object:
   private int[] data becomes private Object[] data

3. Change type of underlying element to T:
   public boolean remove (int target) becomes public boolean remove (T target)
Converting a collection class to a generic collection class

4. Change static methods to generic static methods:
   public static Bag union (Bag b1, Bag b2) becomes
   public static <T> Bag<T> union (Bag<T> b1, Bag<T> b2);

5. Use typecast when retrieving array elements:
   int item = data[itemIndex]; becomes
   T item = (T)data[itemIndex];
Converting a collection class to a generic collection class

6. Use .equals() or .compareTo() instead of relational operators for comparison operations

7. Decide whether or not “null” is a value; if it is, revise rule 5, above – for example, the countOccurences method should be able to count null references as well as actual values, and for null, you would use ==
Converting a collection class to a generic collection class

8. Set unused reference variables to null: in the Bag, this is part of the revised remove method

9. Update documentation to inform clients that your collection is a collection of references to objects
Deep cloning of object collections

• Problem with cloning references to references: since you’re copying a reference, both original and copy point to same data – this is known as a shallow copy

• Deep cloning requires an extra step: check each individual reference to ensure it can be cloned, then give a reference to the original reference’s clone to the copy
It gets worse ...

• The obvious solution would be to clone each data item in the collection’s array, as follows:
  
  ```java
  copy[x] = data[x].clone();
  ```

• Unfortunately, this won’t work: the Object class’s clone() method (remember, all items in data are Objects) isn’t public

• One solution is shown on the next slide
Deep cloning of an Object array

```java
java.lang.Class myClass;
java.lang.reflect.Method myCloneMethod;
try {
    myClass = data[x].getClass();
    myCloneMethod = myClass.getMethod("clone", new Class[0]);
    copy.data[x] = myCloneMethod.invoke(data[x], new Object[0]);
} catch (Exception e) {
    // do shallow clone, as deep clone isn’t available
}
```
Generic Node class

Original Node class
public class IntNode
{
    private int data;
    IntNode link;
    ...
}

Generic version:
public class Node <E>
{
    private E data;
    Node<E> link;
    ...
}
Example method from generic Node class

```java
public static <E> Node<E> listSearch (Node<E> head, E target) {
    Node<E> cursor;
    if (target == null) {
        for (cursor = head; cursor !=null; cursor=cursor.link)
            if (cursor.data == null)
                return cursor;
    }
    else {
        for (cursor=head; cursor!=null; cursor=cursor.link)
            if(target.equals(cursor.data))
                return cursor;
    }
    return null;
}
```
Interfaces

• Interface: list of method signatures without implementation

• Classes can implement interfaces – means implementing class must provide definitions for the methods specified in the interface
Interface advantages

• Signal to client programmer that your class is consistent with others that implement the same interface

• Supports generic programming:
  – client program can be written with references to generic interface
  – objects of implementing class can be assigned to such references
Generic interfaces

• List of related method specifications
• Depend on existence of one or more unspecified (generic) classes
• Java 5.0 provides an example of such an interface: `Iterator<E>`
• Recall that an iterator is a set of methods for stepping through a sequence of elements in a collection
Iterator<T> interface

• Implementing class must:
  – be a generic class with its own generic type parameter
  – provide definitions for 3 methods:
    • public boolean hasNext()
    • public T next()
    • public void remove()
Iterator<T> methods work together

• Call hasNext() method to determine if an element exists at the current position
• If hasNext returns true, can call next(), which returns the current element
• The remove() method removes the element at the current position (provided such an element exists) unless written to throw UnsupportedOperationException
Lister class: external iterator for a linked list

• Requires existence of generic Node class (just as our implementation of Sequence does)
• For Sequence, we relied on internal iterator: methods next, hasNext and start
• External iterator more versatile:
  – can start anywhere
  – can have multiple iterators operating on same list at same time
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Lister<E> implements Iterator<E> {
    private Node<E> list;
    public Lister(Node<E> head) {
        list = head;
    }

    public boolean hasNext() {
        return (list != null);
    }
}

public E next() {
    E answer;
    if (!hasNext())
        throw new NoSuchElementException
            ("The Lister is empty");

    answer = list.getData();
    list = list.getLink();
    return answer;
}

public void remove() {
    throw new UnsupportedOperationException
        ("Lister.remove not allowed");
}
} // end of class
Comparable<T> interface

• Not an empty interface, but requires implementation of only one method:
  public int compareTo(T obj)

• The purpose of the interface is to provide a mechanism for comparing two objects beyond the equals() method provided in the Object class and frequently overridden

• Many of the API classes implement this interface; for example, all of the wrapper classes have headings like:
  public class Integer implements Comparable<Integer>
Testing value for interface compatibility

• If an object’s class implements Comparable, then you know you can use the compareTo method on the object

• If you don’t know for sure, you can use the instanceof operator to determine whether or not your object is Comparable:
  
  SomeType object1, object2;
  ...
  if (object1 instanceof Comparable)
  {
    if (object1.compareTo(object2) > 0)
        ....
Interfaces as parameters

- Can use an interface type to specify parameter for a method
- Since there is no such thing as an actual instance of an interface, need to pass actual argument that passes the “instanceof” test – in other words, an object of a type that implements the interface
- Generic method can use generic interface type parameter
Cloneable interface

- This one is empty: its only purpose is to flag a class (and its instances) so that the clone() method can check its cloneable status (or lack thereof)

- Example:

```java
if (someObject instanceof Cloneable)
    copy = someObject.clone();
else // not cloneable, do shallow copy
    copy = someObject;
```
Iterable interface

• Interface that signals the presence of an Iterator; one required method:
  
  public Iterator<E> iterator()
  
  – returns iterator to be used on collection

• Purpose is to make available a stylized version of a for loop specifically for use with collection classes

• General form of for loop:
  
  for (CollectionType item : collectionObjectName)
  
  – loop starts at front of collection, uses iterator provided by iterator() method to move through to end
Bag class with Iterator

• Can use basically same code as last Bag we looked at (linked list based), but add a couple of things

• In class heading:
  
  ```java
  public class LinkedBag<E>
  implements Cloneable, Iterable {
  
  public Iterator<E> iterator () {
      return (Iterator<E>) new Lister<E>(head);
  }
  ```

• Add iterator method:
  
  ```java
  public Iterator<E> iterator () {
      return (Iterator<E>) new Lister<E>(head);
  }
  ```
Collections Framework

• Java API contains library of useful data structures
• Collections library also serves as framework for adding new collection classes that can interact with existing classes
Overview of collections framework

• Collection: data structure that contains multiple objects (elements)

• Collections framework specifies four interfaces:
  – Collection: general collection (bag)
  – Set: unordered collection of unique elements
  – SortedSet: ordered collection of unique elements
  – List: ordered collection of elements; can contain duplicates
Concrete classes supplied by collections framework

• HashSet: a set implementation that uses hashing to locate the set elements
• TreeSet: a sorted set implementation that stores the elements in a balanced binary tree
• LinkedList and ArrayList: two implementations of the List interface type
Collection and Iterator

• Collection and Iterator are the fundamental interfaces of the collections framework
• A collection is any class that can store multiple elements; individual collection classes enforce different rules for how data are to be stored and located
• An iterator is a mechanism for visiting all elements in a collection
Methods specified by Collection interface

boolean add(Object obj)  boolean isEmpty()
boolean addAll(Collection c)  Iterator iterator()
void clear()  boolean remove(Object obj)
boolean contains(Object obj)  boolean removeAll(Collection c)
boolean containsAll(Collection c)  boolean retainAll(Collection c)
boolean equals(Object obj)  int size()
int hashCode()  Object[] toArray()
Object[] toArray(Object[] a)
AbstractCollection class

- Collection is a hefty interface: client programmer must implement 15 methods
- AbstractCollection class relieves client programmer of this burden; provides reasonable default implementations for almost all of these methods
Example method from AbstractCollection

```java
public Object[] toArray()
{
    Object[] result = new Object[size()];
    Iterator e = iterator();
    for (int i=0; e.hasNext(); i++)
        result[i] = e.next();
    return result;
}
```

Note use of size() and iterator(); these are the two methods left undefined by the abstract class.
Set Interface

• Definition of Set:
  
  public interface Set extends Collection{}

• No methods - why have it?
  - Set is conceptually a subtype of Collection; Set is unordered Collection without duplicates
  - Separate interface serves to tag objects with Set quality; algorithms can require Set objects as distinct from other Collections
List interface

• List: ordered collection of elements
• Each list position accessible via integer index
• Interface specifies several additional methods to general Collection type; most of these are concerned with index positions
List interface methods

boolean add(int index, Object obj)
boolean addAll(int index, Collection c)
Object get(int index)
int indexOf(Object obj)
int lastIndexOf(Object obj)
ListIterator listIterator()
ListIterator listIterator(int index)
Object remove(int index)
Object set(int index, int Object)
List subList(int fromIndex, int toIndex)
ListIterator interface

• Subtype of Iterator
• Adds:
  – support for indexing
  – ability to move both forward and backward through list
ListIterator interface

- Methods include:
  - int nextIndex()
  - int previousIndex()
  - boolean hasPrevious()
  - Object previous()
  - void add(Object obj)
  - void set(Object obj)
List classes

- ArrayList
- LinkedList
  - provides “indexed” access to individual list elements, but it’s klunky and slow (must visit all predecessor elements to get to desired element)
  - highlights weakness in the framework design; should have provided separate interface types for indexed collections (arrays) and ordered collections (lists)
RandomAccess interface

- Version 1.4 of Java sdk introduced RandomAccess interface to fix design problem
- Tagging interface (no methods) - can use to test whether or not indexed access is really appropriate for a particular object
- ArrayList implements RandomAccess, but LinkedList does not
Optional operations

• API documentation tags some methods as optional operations
• Default implementations throw UnsupportedOperationException
• Collection methods add() and remove() are examples
Views

- View: object that implements an interface of collections framework, but permits only restricted access to data structure
- Optional operations exist to support views
- Built-in Java type array has no methods; can apply view to enhance array functionality
Views

• Example: asList method
  – asList turns an array into a view object: collection that implements List interface, so List methods can be applied to array elements
  – does not copy array elements; get and set methods of view object access original array
  – Can’t apply add() or remove() methods to view, because size of underlying array can’t be changed: this is why these methods are “optional”
API classes implementing `Collection<E>`

- **Vector**: like an array, but grows automatically as needed
- **Set**: like a Bag, in that order is not required, but unlike a Bag, uniqueness is required
- **List**: ordered collection (like a Sequence)
- **ArrayList**: List with indexing added
- **SortedSet**: Requires both uniqueness and ordering
- **HashSet**: generic implementation of a hash table