• Java Generics
• The Java Type System and Type Wildcards
• Copying Java Objects
• Deep vs. Shallow Copy
• The Cloneable interface and the clone() method
Java Generics and the Java Type System

- Consider the inheritance hierarchy shown in this UML diagram:

- According to this hierarchy, every Circle is a Shape

- But, is a Collection<Circle> a Collection<Shape>?
Java Generics and the Java Type System

```java
Collection<Shape> cs;
Collection<Circle> cc = new LinkedList<Circle>();
cs = cc;  // compile-time error!
```

- Java parameterized types are *not* covariant:

- Even though every Circle is a Shape, the Java type system does *not* consider every Collection<Circle> to be a Collection<Shape>

- Why not?
Java parameterized types are not covariant

• If D is a subtype of B, then every behavior of B is also a behavior of D
  – This is a basic aspect of the “is-a” relationship: every D is-a B

• So if Collection<Circle> is a subtype of Collection<Shape>, then any behavior of a Collection<Shape> is also a behavior of Collection<Circle>.

• But a Collection<Shape> has the behavior that you can add a Rectangle to it, and this is not a behavior of Collection<Circle>!
Goal: Create a method to display all the Shapes in a collection.
Problem: How to express the type of the method parameter?

```java
// Display all Shape objects in the given Collection.
// Call their display() instance method to do that.
static void displayShapes( Collection<Shape> collection ){
    for ( Shape shape : collection )
        shape.display() ;
}

Collection<Shape> shapes = new LinkedList<Shape>();
shapes.add( new Circle( 5.0 ) );
shapes.add( new Rectangle( 4.5, 21.2 ) );
displayShapes( shapes );

Collection<Circle> circles = new LinkedList<Circle>();
circles.add( new Circle( 5.0 ) );
circles.add( new Circle( 15.0 ) );
circles.add( new Circle( 25.0 ) );
displayShapes( circles );        // ERROR!
```
Java Generics and the Java Type System

```java
Collection<Shape> cs;
Collection<Circle> cc = new LinkedList<Circle>();
cs = cc;  // compile-time error!
```

- Java parameterized types are not covariant
- Even though every Circle is a Shape, the Java type system does not consider every Collection<Circle> to be a Collection<Shape>
But suppose we want to write a method that can take as argument a Collection<X> for any type X that is a Shape… How can we do that?

We somehow need more flexibility in the type parameter

That is, instead of having to say “Collection<X>” for a specific type X, we want to be able to say “Collection<X>, for any X in a range of types”.

Java’s solution to this uses the type wildcard: ?
The unbounded wildcard

The wildcard ? used by itself is called an unbounded wildcard.

<?> means: any type

Thus Collection<?> is a “collection of any type.”

```java
18 // A most general print method. It will print collections of
19 // any kind of type. Note the variable type in the for loop.
20 static void printCollection( Collection<?> collection ){
21    for ( Object element : collection )
22        System.out.println( element );
23 }
```
Unbounded wildcard example: `Collection.containsAll()`

`Collection<E>` requires an instance method

`containsAll( Collection<?> c )`

Here’s one way to implement it:

```java
public boolean containsAll( Collection<?> c ) {
    Iterator<?> e = c.iterator();
    while ( e.hasNext() )
        // does this collection contain the
        // next element from c?
        if( !contains( e.next() ) )
            // nope, c has an element we don’t have
            return false;
    return true; // yep, we have all the elements c has
}
```

Say in English what `containsAll( Collection<?> c )` does...
Bounded Wildcards

- But sometimes you don’t want the broad inclusiveness of the unbounded wildcard…
- … instead you want to put a *bound* on the range of types accepted.
- The *bounded wildcard* does this. There are two forms:

  `<? extends T>` means: “T, or any type that is a descendant of T”
  `<? super T>` means: “T, or any type that is an ancestor of T”

- (So, `<? extends Object>` is equivalent to `<?>` ;
  `<? super Object>` is equivalent to `<Object>` )

- Let's consider another example from the JCF…
Consider `AbstractCollection`:

```java
public abstract class AbstractCollection<E> implements Collection<E> {

    // Add the element e to this Collection
    public boolean add(E e) {

        // This means we can use the add() method to store
        // an element of type E or any of E's subclasses
        // in the collection: you can put into this collection
        // anything that “is-a” E
    }
```
Bounded Wildcard

• AbstractCollection also needs a method `addAll()` which takes a collection, and adds all the elements of that collection to this collection

```java
public abstract class AbstractCollection<E> implements Collection<E> {

    // Add all the elements of the argument Collection
    // to this Collection
    public boolean addAll( ... ) {
```

• Question: What should be the type of the argument to `addAll()`?
Bounded Wildcard

public abstract class AbstractCollection<E> implements Collection<E> {

What should be the parameter for `addAll()`?

1st attempt

```
public boolean addAll(Collection<E> c)
```

Too restrictive. This would preclude `addAll`ing a Collection<Circle> to a Collection<Shape> or a Collection<Integer> to a Collection<Number>.

2nd attempt

```
public boolean addAll(Collection<?> c)
```

Not restrictive enough. Would let you try to `addAll` a Collection<Shape> to a Collection<Number>. 
Bounded Wildcard: `<? extends E>`

What we need is a flexible mechanism that will allow us to specify a *range* of types *constrained* by an “*upper bound*” on the type range. Java’s **bounded wildcard** does this.

```java
1  // method from AbstractCollection<E>
2  public boolean addAll(Collection<? extends E> c) {
3      boolean modified = false;
4      Iterator<? extends E> e = c.iterator();
5
6      while ( e.hasNext() ) {
7          if ( add( e.next() ) )
8              modified = true;
9      }
10     return modified;
11 }
```

The bounded wildcard in the parameter type means that the `Collection` type is *not exactly known*, but is bounded above by type `E`. That is, the element type of `c` must be `E` or one of `E`’s subclasses.
Solving the \texttt{displayShapes()} problem

- An upper-bounded wildcard is exactly the solution we need to define a method that can take as argument a Collection of any kind of Shape:

```java
18  // \textit{A method that will display collections of}
19  // \textit{any kind of Shape. Note the variable type in the for loop.}
20 static void displayShapes( Collection<? extends Shape> collection ){
21      for ( Shape shape : collection )
22          shape.display() ;
23  }
```
Java Generics: Displaying a Collection

**Goal:** Create a method to display all the Shapes in a collection.

**Solution:** Use a bounded wildcard!

```java
// Display all Shape objects in the given Collection.
// Call their display() instance method to do that.
static void displayShapes( Collection<? extends Shape> collection ){
    for ( Shape shape : collection )
        shape.display();
}

Collection<Shape> shapes = new LinkedList<Shape>();
shapes.add( new Circle( 5.0 ) );
shapes.add( new Rectangle( 4.5, 21.2 ) );
displayShapes( shapes );

Collection<Circle> circles = new LinkedList<Circle>();
circles.add( new Circle( 5.0 ) );
circles.add( new Circle( 15.0 ) );
circles.add( new Circle( 25.0 ) );
displayShapes( circles ); // OK!
```
Generic Methods

• Just as classes can be generic, so can methods

```java
public <T> Collection<T> copy( Collection<T> c )
```

Declaration

```
Collection<String> pets = new Collection<String>();

...  
```

Application

```
Collection<String> petsCopy = copy( pets );
```

<table>
<thead>
<tr>
<th>type variable (type parameter)</th>
<th>use of type variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>formal type parameter list</td>
<td>actual type to map to T is inferred from the element type of pets</td>
</tr>
</tbody>
</table>
Generic method: an example

• The Collection<E> interface requires this method:
  public <T> T[] toArray(T[] a)

• The documentation says:

  Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. If the collection fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this collection.
Another toArray() method

- Compare to the `public Object[] toArray()` method of Collection, which always allocates and returns a new array of type `Object[]`:

  Returns an array containing all of the elements in this collection. The returned array will be "safe" in that no references to it are maintained by this collection. (In other words, this method must allocate a new array even if this collection is backed by an array).

- Note: the returned array will be a new array, but it will contain references to the same `data elements` as those contained in the collection!
Copying Collections

• The toArray() methods of Collection permit creating an 'array equivalent' of a collection
• This can be useful, since some implementations of algorithms operate on arrays, not Collection objects
• However sometimes it is useful to create a copy of a Collection, as another Collection
• Java provides various mechanisms to do this
• But first doing it requires understanding the difference between different ‘depths’ of copy...
How deep is the copy?

• What does it mean to *copy* a data structure?
  You might create a copy of a pointer pointing to an existing data structure object (very shallow copy)
• You might create a new data structure object, containing copies of the existing data structure's instance variables (shallow copy)
• You might also create copies of objects the instance variables point to (deep copy)
• You might also create copies of objects those objects' instance variables point to (deeper copy)... etc., etc.
Very shallow copy

- If you just copy the pointer to the data structure object, you just have created another way of referring to one and the same object.

- So for example now x.clear() or y.clear() do the same thing.
Shallow copy

• If you create a new data structure object and copy its instance variables you are still sharing structure

• This can lead to problems: for example x.clear() now puts the object y points to in an inconsistent state!
Deep copy

- Create a new data structure object, and also new data structure elements, but share data item objects.

- Note: If the data item objects are mutable, changing their values will change them in both data structures.
Deeper copy

- Create a new data structure object, and also new data structure elements, and also copy data item objects

- Now you have two completely independent copies (but what if the data objects have pointers to other objects?...)

![Diagram with data structure objects and their values]
If you want to design instances of a class so that they are copyable, how can you do that? There are several options:

1. Define a copy constructor for your class: a constructor that takes as argument a pointer to an existing instance of the class, and initializes the new instance to be a copy of the existing one.

2. Define a static factory method in the class that takes as argument a pointer to an existing instance of the class, creates a new instance, and initializes the new instance to be a copy of the existing one.

3. Use the Cloneable interface and override the clone() instance method.
Copying objects in Java: the `clone()` method

- This method is defined in the Object class:
  \[\text{protected Object clone()}\]
- As defined in the Object class, this method does shallow copy: it makes a byte-by-byte copy of the object
- Note it has protected visibility!
- To make instances of a class copyable by calling a public `clone()` method, do the following:
  - declare the class to implement the Cloneable interface,
  - override Object's `clone()` method, giving it public visibility
How deep a copy does the `clone()` method do?

• The `clone()` method inherited from `Object` does shallow copy
• Note: Concrete Java Collection classes such as `LinkedList` override `clone()` to do what we called “deep copy”: structure is copied, data objects are shared
• It is difficult in general to do what we called “deeper copy”, because the data objects may not implement `Cloneable`!
• In general, any copying technique can be defined to do shallow or deep copy. Document it accordingly
Next time

• What is recursion?
• Recursive definitions
• Recursive programming
• Relationship between recursion and iteration
• Time and space costs of recursion

Reading: Gray, Ch 8