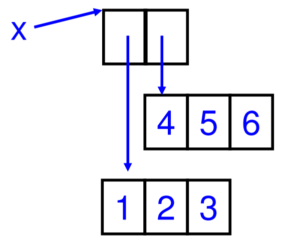
CMSC 330: Organization of Programming Languages

Generics and Polymorphism, con't.

Arrays in Java

- In Java, arrays are objects, and therefore are subclasses of Object
- Multidimensional Java arrays are therefore arrays of objects
  - int[][] x = {{1, 2, 3}, {4, 5, 6}};



- Comparison to C?
  - More uniform
  - Requires more memory (for pointers)
  - Requires two dereferences to access an element

Subtyping and Arrays

- Java has one funny subtyping feature:
  - If S is a subtype of T, then S[] is a subtype of T[]
- This lets us write methods that take arbitrary arrays

```
public static void reverseArray(Object[] a) {
  for(int i= 0, j= a.length - 1; i < j; i++, j--) {
    Object tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
  }
}
```

Problem with Subtyping Arrays

```
public class A { ... }
public class B extends A { void newMethod(); }
...
  void f(void) {
     B[] bs = new B[3];
     A[] as;

     as = bs;            // since B[] subtype of A[]
     as[0] = new A();    // (1)
     bs[0].newMethod(); // (2)
  }
```

- Program compiles without warning
- Java must generate a runtime check at (1), that the type written to an array element is a subtype of the array's contents (which it's not in this case), to prevent (2)

Subtyping for Generics

- Is Stack<Integer> a subtype of Stack<Object>?
  - We could have the same problem as with arrays
  - Java forbids this case at compile time
- But what do we do if we have a method that can operate generically on a parameterized type?

```
int count(Collection<Object> c) {
  int j = 0;
  Iterator<Object> iter = c.iterator();
  while (iter.hasNext()) {
    Object e = iter.next();
    j++;
  }
  return j;
}
```

Solution #1: Use Polymorphic Methods

```
<T> int count(Collection<T> c) {
  int j = 0;
  Iterator<T> iter= c.iterator();
  while (iter.hasNext()) {
    T e = iter.next();
    j++;
  }
  return j;
}
```

- But requires a "dummy" type variable that isn't really used for anything

Solution #2: Wildcards

- Use ? as the type variable- Collection<?> is "Collection of unknown"

```
int count(Collection<?> c) {
  int j = 0;
  Iterator<Object> iter = c.iterator();
  while (iter.hasNext()) {
    Object e = iter.next();
    j++;
  }
  return j;
}
```

- Why is this safe?
  - Using ? is a contract that you'll never rely on having a particular parameter type
  - All objects are subtypes of Object, so assignment to e ok

Legal Wildcard Usage

- Reasonable question:
  - Stack<Integer> is not a subtype of Stack<Object>
  - Why is Stack<Integer> a subtype of Collection<?>?
- Answer:
  - Wildcards permit "reading" but not "writing"

Example: Can Read But Cannot Write c

```
int count(Collection<?> c) {
  int j = 0;
  Iterator<?> iter = c.iterator();
  while (iter.hasNext()) {
    Object e = i.next();
    c.add(e);  // fails: Object is not ?
    j++;
  }
  return j;
}
```

More on Generic Classes

- Suppose we have classes Circle, Square, and Rectangle, all subtypes of Shape

```
void drawAll(Collection<Shape> c) {
  for (Shape s : c)
    s.draw();
}
```

  – Can we pass this method a Collection<Square>?
    - No, it's not a subtype of Collection<Shape>
  – How about the following?

```
void drawAll(Collection<?> c) {
  for (Shape s : c) // not allowed
    s.draw();
}
```

Bounded Wildcards

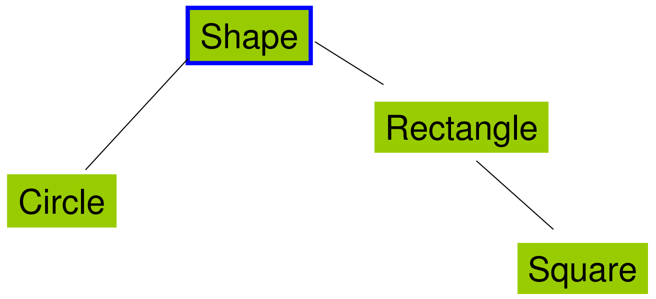
- We want drawAll to take a Collection of anything that is a *subtype* of shape

```
void drawAll(Collection<? extends Shape> c) {
  for (Shape s : c)
    s.draw();
}
```

  – This is a *bounded wildcard*
  – We can pass Collection<Circle>
  – We can safely treat e as a Shape

Upper Bounded Wildcards

- ? extends Shape actually gives an *upper bound* on the type accepted
- Shape is the upper bound of the wildcard



Bounded Wildcards, con't.

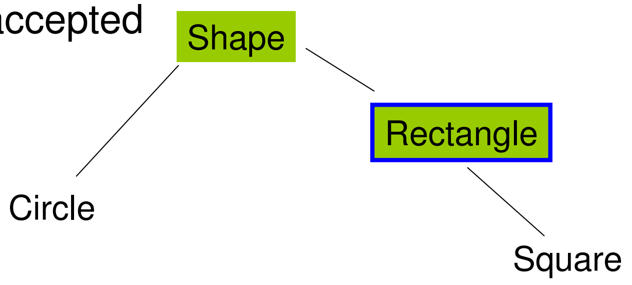
- Should the following be allowed?

```
void f(Collection<? extends Shape> c) {
  c.add(new Circle());
}
```

  - No, because c might be a Collection of something that is not compatible with Circle
  - This code is forbidden at compile time

Lower Bounded Wildcards

- Dual of upper bounded wildcards
- ? super Rectangle denotes a type that is a supertype of Rectangle
  - Type Rectangle is included
- ? super Rectangle gives a *lower bound* on the type accepted



Lower Bounded Wildcards, con't.

- Now the following is allowed

```
void f(Collection<? super Circle> c) {
  c.add(new Circle());
  c.add(new Rectangle());  // fails
}
```

  - Because c is a Collection of something that is always compatible with Circle

Bounded Type Variables

- You can also add bounds to regular type variables

```
<T extends Shape> T getAndDrawShape(List<T> c) {
  c.get(1).draw();
  return c.get(2);
}
```

  - This method can take a List of any subclass of Shape