LearnQuest

Presents

OO Programming with Java

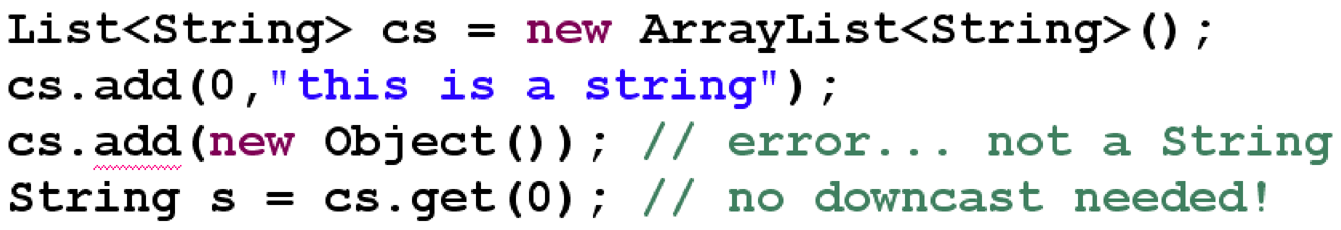
Generics

Learning Objectives

- Understand what generics are and why they are needed
- Learn how generics are declared
- See the different types of generics – wildcards, sub-typing and super-types
- Show how generics are used in method declarations
- Learn about generic erase and transition

What are Generics?

- Generics allow “abstraction over types.”
  - ✓ You can consider this a fancy way of saying that we can write generic code without losing type safety
  - ✓ Generics provide a way to declare the types of objects used in a context by generic code, such as a container
  - ✓ Allows compile-time type checking and helps (greatly) to avoid runtime typecasting exceptions
  - ✓ Reduces (almost eliminates) the need for down-casting
  - ✓ Widely used in the Collection Framework



```
List<String> cs = new ArrayList<String>();
cs.add(0,"this is a string");
cs.add(new Object()); // error... not a String
String s = cs.get(0); // no downcast needed!
```

Defining Simple Generics

- Generics are defined using a declaration of *formal type parameters* using angle brackets.

```
public interface List<E> {
    void add(E x);
    Iterator<E> interator();
}
```

- When used, the *parameterized type* (the contents of the angle brackets) specify the type for that specific use …

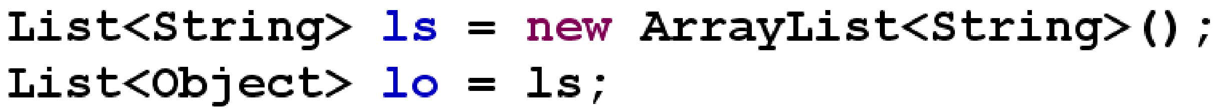
```
public class ListTest {

    List<String> ls;
```

Generic Compilation

- Note that the declaration of a generic is compiled once and only once.
  - ✓ When a generic is invoked, the actual type is replaced by the arguments used for that specific invocation.
- You can think of generics working much like a method…
  - ✓ When invoked, the formal method value parameters are replaced by the actual values passed at runtime.

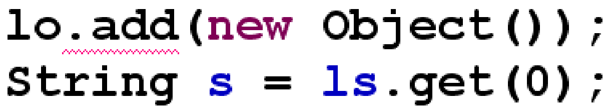
Generic Subtyping

- Is a List of Strings the same as a List of Objects?



```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

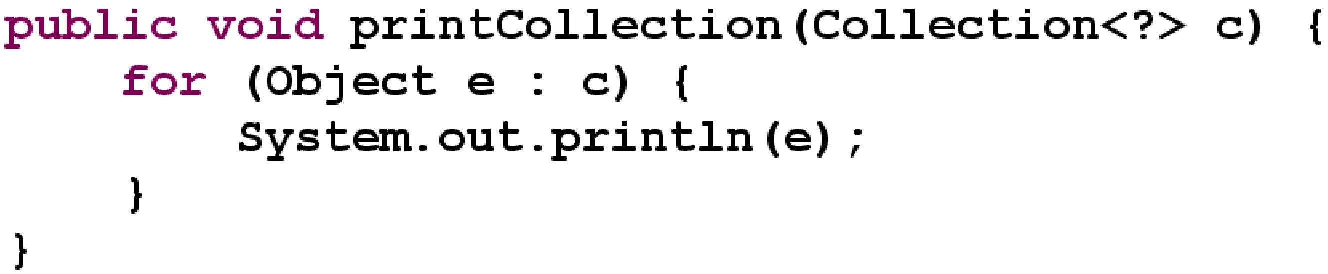
- The answer is no… in the above code line 1 is fine, but line 2 will cause a compile error.
- If a class X has a subtype Xsub and we make a generic declaration E<X> and E<Xsub>, E<Xsub> is NOT a subtype of E<X>.
  - ✓ This safety mechanism prevents runtime errors such as:



```
lo.add(new Object());
String s = ls.get(0);
```

Wildcards

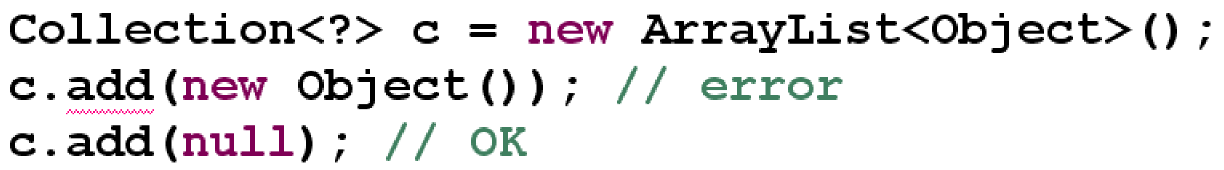
- Generics provide a type that can be used as a placeholder for *all* possible types.
  - ✓ This is called a *wildcard type*
  - ✓ The notation used is the question mark “?”
- The following shows the use of the wildcard to specify that the Collection passed may contain *any* type of object:



```
public void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Wildcards

- It is not safe to try to add an object to a container element declared using the wildcard.
  - ✓ The ? denotes an unknown type – the only valid entry is null. See below.

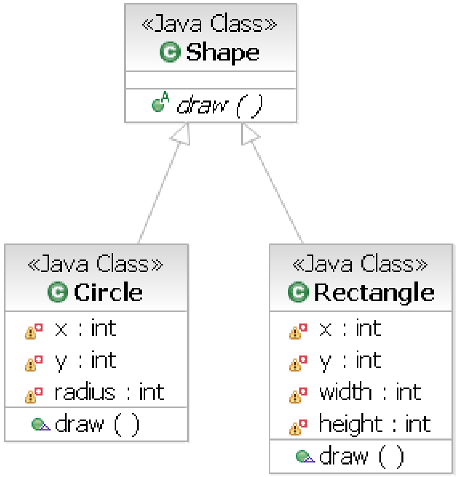


```
Collection<?> c = new ArrayList<Object>();
c.add(new Object()); // error
c.add(null); // OK
```

Bounded Wildcards

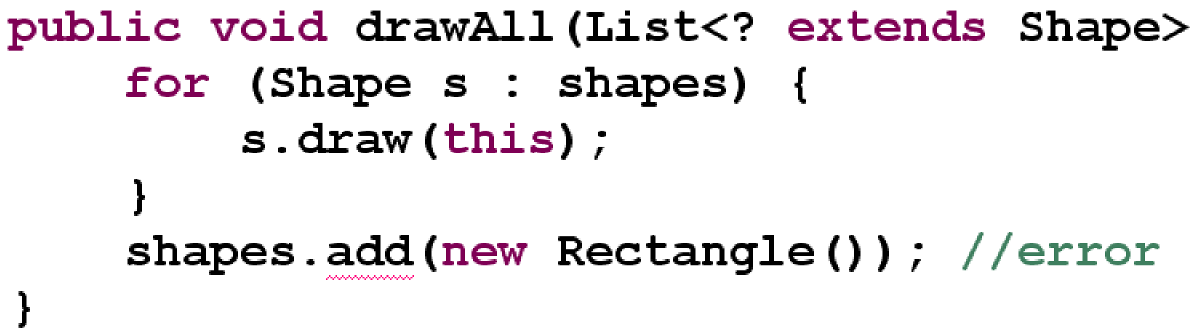
- Bounded wildcards make use of the unknown type symbol and add the *extends* keyword to specify an upper bound to the wildcard.
  - ✓ Objects in the collection must be a subtype of the upper bound.

```
public void drawAll(List<? extends Shape> shapes){
     for (Shape s : shapes) {
          s.draw(this);
     }
}
```



Bounded Wildcard Constraints

- Remember that ? designates an unknown type. This means that the following code is illegal:

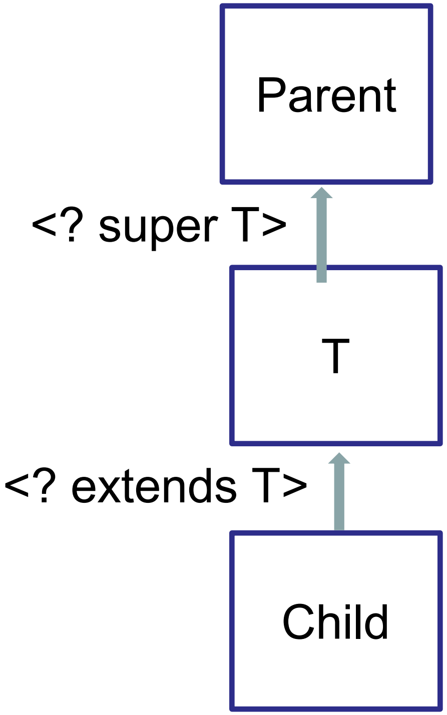


```
public void drawAll(List<? extends Shape> shapes){
     for (Shape s : shapes) {
          s.draw(this);
     }
     shapes.add(new Rectangle()); //error
}
```

- The reason is that we don’t know specifically which subclass of shape the list is supposed to hold, just that it is some shape.

Bounded Wildcard Constraints

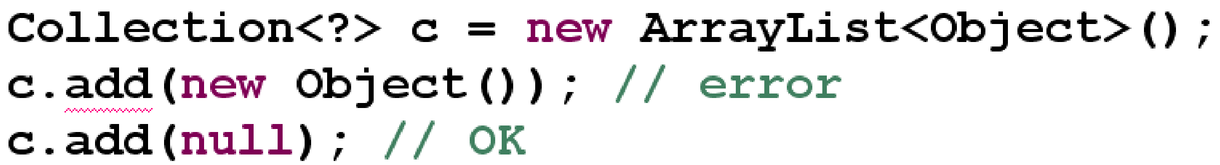
- The problem of not being able to add when using <? extends T> might be easier to understand if we recognize how it relates to inheritance.
- The declaration <? extends T> means that the wildcard matches T or some subclass of T, but we don't know which.
- Conversely, <? super T> means that the wildcard matches T or some superclass of T, but we don't know which.



LearnQuest

Consequences of Bounded Wildcards

- A consequence of these rules is that when using bounded wildcards:
  - ✓ Using <? extends T> allows using the references, but not adding or assigning, except for **null**, because we don’t know the exact type allowed.
    - This also applies to <?> for the same reason, meaning that this code is illegal, except the unique case of **null**:



```
Collection<?> c = new ArrayList<Object>();
c.add(new Object()); // error
c.add(null); // OK
```

  - ✓ Using <? super T> allows adding or assigning, since a superclass reference can always refer to a subclass, but we can’t use the references because, again, we do not know the exact type.

Generic Methods

- Individual methods can also declare type parameters. The syntax is similar to that we've used with types:

  access-modifiers type-parameters return-type name(args)

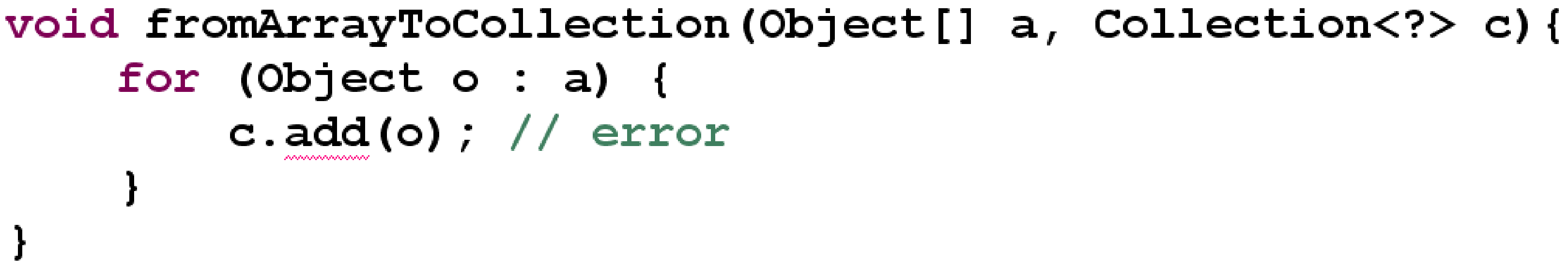
- For example:

```
public static <T> Collection<T> unmodifiable(Collection<T> c);
public <T> T getBean(Class<T> requiredType);
```

- Even the return type can be influenced by the type parameters. The second example is from Spring, where `getBean` declares that it will return only a bean of the type whose class you pass in, with no need to cast.

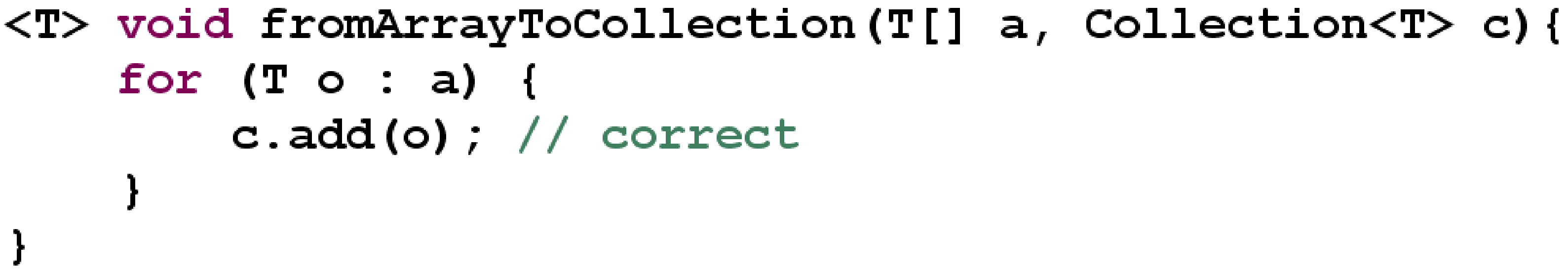
Example Generic Method

- We want to write a method that takes an array of objects and places that array into a Collection…



```
void fromArrayToCollection(Object[] a, Collection<?> c){
    for (Object o : a) {
        c.add(o); // error
    }
}
```

- A solution is to use a generic method - a parameterized declaration of the arguments passed. The type will be derived and enforced.



```
<T> void fromArrayToCollection(T[] a, Collection<T> c){
    for (T o : a) {
        c.add(o); // correct
    }
}
```

Generic Method Argument Types

- The argument types passed to a generic method are inferred by the compiler based on the types of the actual arguments.
  - ✓ The compiler will infer the most specific type that will make the method invocation type-correct.
- Therefore, use generic methods when there are dependencies between the arguments being passed to a method and/or the return type.
- Whereas, use wildcards for the reasons we’ve explored, such as helping to maintain polymorphism, as with Collections of bounded types
  - ✓ Wildcards are used to support flexible sub-typing.

A Final Word on Bounded Wildcards

- Do you recall our earlier example where SportsCar extends Car and implements Convertible, and SUV extends Cargo and implements Convertible?
- How can we describe a method such that it accepts only types that are Car **and** Convertible?

```
public static <T extends Car & Convertible> park(T car) {
        car.setSpeed(0);
        car.putTopUp();
}
```

- This defines T as a type that both extends Car **and** Convertible.
  - ✓ You can add as many interfaces as you want, but the superclass must come first, right after the `extends` keyword.

Using Generics with Legacy Code

- When a generic type (like Collection) is used without type parameters it is called a *raw type*.

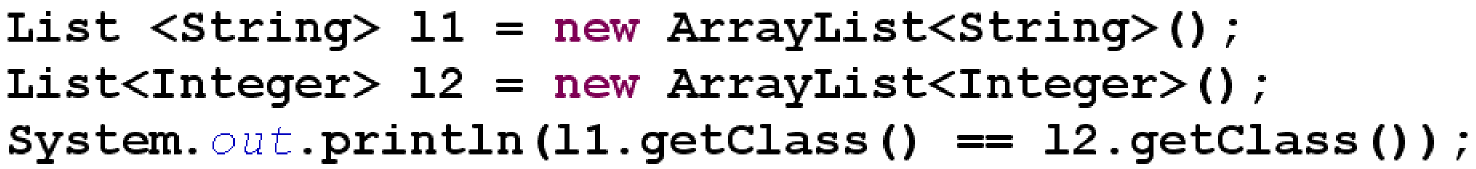
  - ✓ Use of raw types generate unchecked warnings by the compiler.
  - ✓ Most IDEs provides a *quick fix* for raw types.
  - ✓ There are no compile-time guarantees that a method invocation on a raw type will not throw a class cast exception.
  - ✓ It is the responsibility of the programmer to ensure that the code will work as expected.

Type Erasure

- Generics are implemented by the compiler using a conversion process called ***type erasure***.
  - ✓ The compiler converts all generic code into non-generic.
  - ✓ All type information between the angle brackets is removed.
  - ✓ All type variables are replaced by the upper bound of the type variable (usually Object).
  - ✓ Casts are added as necessary to re-type the objects as needed.
- Type erasure allows generic code to be used with libraries that were created prior to Java 5.

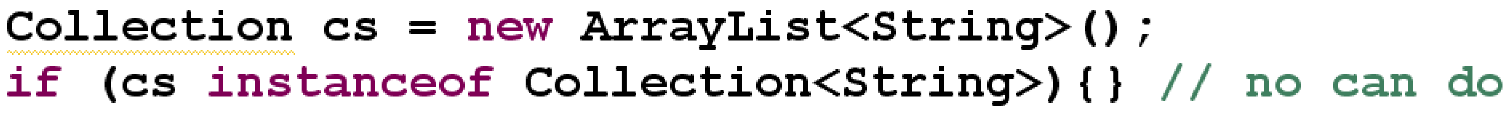
Generic Class Sharing

- The output from the code fragment shown below will be… ***true***.



```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

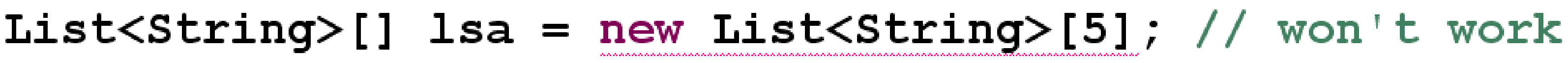
- All generic classes have the same class at runtime (type erasure) regardless of the type parameters.
  - ✓ All static variables and methods of those classes are shared.
  - ✓ This is also why it is illegal to refer to the type parameters of a generic.



```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>){} // no can do
```

Arrays and Generics

- The type of an array may not be a generic type other than a wildcard type…



```
List<String>[] lsa = new List<String>[5]; // won't work
```

- You may create arrays that contain objects that are generic types..

Java Programming Language Updates

- Type Inference for Generic Instance Creation
  - ✓ In Java SE 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>):
    - Map<String, List<String>> myMap = new HashMap<>();

Summary

- In this unit, we saw:
  - ✓ What generics are and why they are needed
  - ✓ How generics are declared
  - ✓ The different types of generics – wildcards, sub-typing and super-types
  - ✓ How generics are used in method declarations

Questions