GENERICS

Motivating Example – Old Style

The cast is annoying but essential!

```
List stones = new LinkedList();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = (Stone) stones.get(0);
```

```
public int countStones(Color color) {
        int tally = 0;
        Iterator it = stones.iterator();
        while (it.hasNext()) {
                Stone stone = (Stone) it.next();
                if (stone.getColor() == color) {
                        tally++;
                }
        }
        return tally;
}
```

Motivating example – new style using generics

List is a *generic interface* that takes a type as a *parameter.*

```
List<Stone> stones = new LinkedList<Stone>();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = /*no cast*/ stones.get(0);
```

```
public int countStones(Color color) {
        int tally = 0;
        /*no temporary*/
        for (Stone stone : stones) {
                /*no temporary, no cast*/
                if (stone.getColor() == color) {
                        tally++;
                }
        }
        return tally;
}
```

Compile Time vs. Runtime Safety

Old way

```
List stones = new LinkedList();
stones.add("not a stone");

...

Stone stone = (Stone) stones.get(0);
```

No check, unsafe

Runtime error

New way

```
List<Stone> stones = new LinkedList<Stone>();
stones.add("not a stone");

...

Stone stone = stones.get(0);
```

Compile time check

Runtime is safe

Stack Example

```
public interface StackInterface {
    public boolean isEmpty();
    public int size();
    public void push(Object item);
    public Object top();
    public void pop();
}
```

Old way

```
public interface StackInterface<E> {
    public boolean isEmpty();
    public int size();
    public void push(E item);
    public E top();
    public void pop();
}
```

New way:
we define a generic interface that takes a type parameter

Linked Stack Example

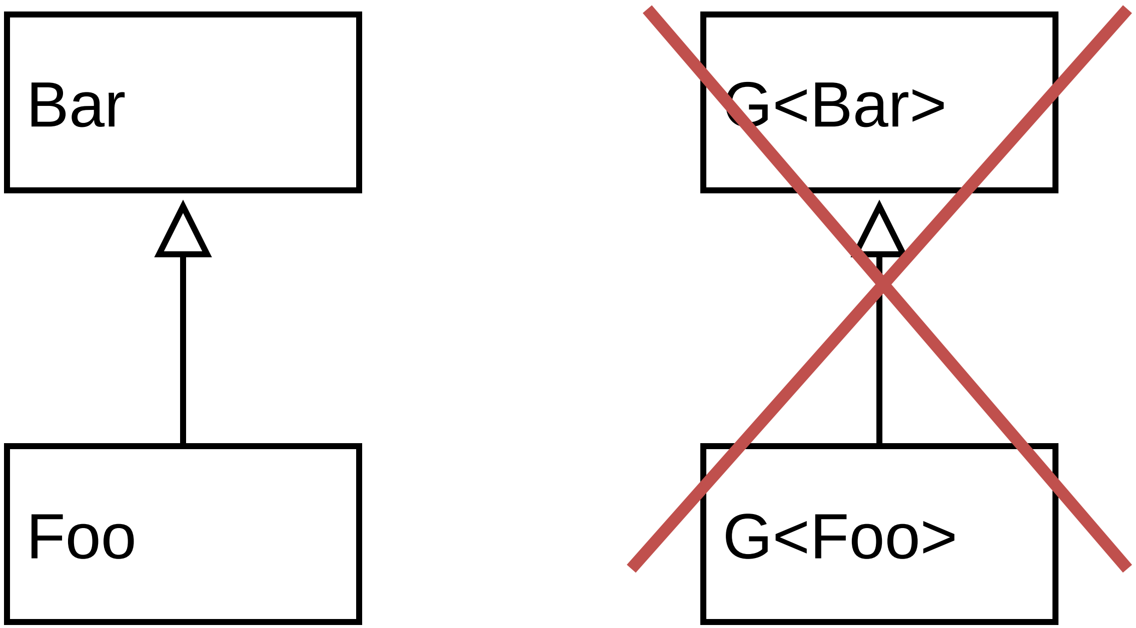
```
public class LinkStack<E> implements StackInterface<E> {
…
    public class Cell {
        public E item;
        public Cell next;
        public Cell(E item, Cell next) {
            this.item = item;
            this.next = next;
        }
    }

…
    public E top() {
        assert !this.isEmpty();
        return top.item;
    }
```

Creating a Stack of Integers

```
Stack<Integer> myStack = new LinkedStack<Integer>();
myStack.push(42); // autoboxing
```

When a generic is instantiated, the *actual type parameters* are substituted for the *formal type parameters*.

Generics and Subtyping



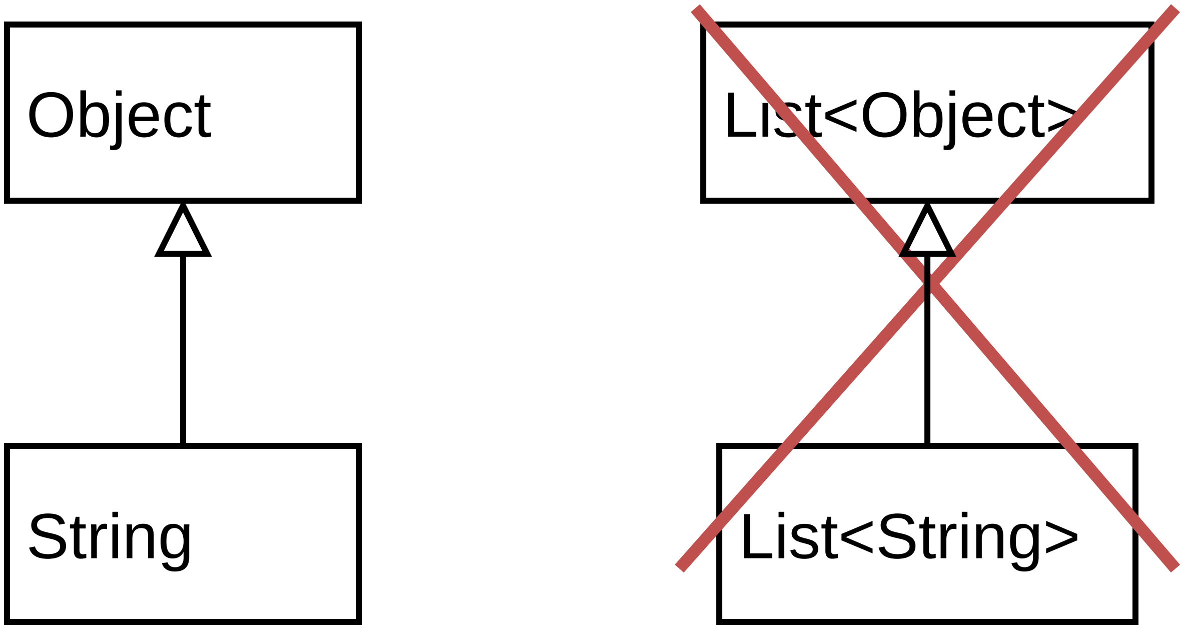
In Java, Foo is a subtype of Bar only if Foo's interface *strictly includes* Bar's interface. Instantiated generics normally have *different* interfaces.

Generics and Subtyping (II)

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

List<Object> lo = ls;
```

Compile error as it is not type safe!

In other words…



A Class Definition with a Type Parameter

Display 14.4 **A Class Definition with a Type Parameter**

```
public class Sample<T>
{
    private T data;

    public void setData(T newData)
    {
        data = newData;
    }

    public T getData()
    {
        return data;
    }
}
```

T is a parameter for a type.

A Class Definition with a Type Parameter (Cont'd)

- A class that is defined with a parameter for a type is called a generic class or a parameterized class
  - The type parameter is included in angular brackets after the class name in the class definition heading.
  - Any non-keyword identifier can be used for the type parameter, but by convention, the parameter starts with an uppercase letter.
  - The type parameter can be used like other types used in the definition of a class.

Generic Class Definition: An Example

Display 14.5 A Generic Ordered Pair Class

```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair()
    {
        first = null;
        second = null;
    }

    public Pair(T firstItem, T secondItem)
    {
        first = firstItem;
        second = secondItem;
    }
    public void setFirst(T newFirst)
    {
        first = newFirst;
    }

    public void setSecond(T newSecond)
    {
        second = newSecond;
    }

    public T getFirst()
    {
        return first;
    }
```

Constructor headings do not include the type parameter in angular brackets.

(continued)

Generic Class Definition: An Example (Cont'd)

Display 14.5 A Generic Ordered Pair Class

```
    public T getSecond()
    {
        return second;
    }

    public String toString()
    {
        return ( "first: " + first.toString() + "\n"
                    + "second: " + second.toString() );
    }

    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            Pair<T> otherPair = (Pair<T>)otherObject;
            return (first.equals(otherPair.first)
               && second.equals(otherPair.second));
        }
    }
}
```

A Generic Constructor Name Has No Type Parameter!!!

- A constructor can use the type parameter as the type for a parameter of the constructor, but in this case, the angular brackets are not used:

```
public Pair(T first, T second)
```

- However, when a generic class is instantiated, the angular brackets are used:

```
Pair<String> pair = new Pair<String>("Happy", "Day");
```

Using Generic Classes and Automatic Boxing

Display 14.7 Using Our Ordered Pair Class and Automatic Boxing

```java
import java.util.Scanner;

public class GenericPairDemo2
{
   public static void main(String[] args)
   {
        Pair<Integer> secretPair =
             new Pair<Integer>(42, 24);

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter two numbers:");
        int n1 = keyboard.nextInt();
        int n2 = keyboard.nextInt();
        Pair<Integer> inputPair =
            new Pair<Integer>(n1, n2);

        if (inputPair.equals(secretPair))
        {
            System.out.println("You guessed the secret numbers");
            System.out.println("in the correct order!");
        }
        else
        {
            System.out.println("You guessed incorrectly.");
            System.out.println("You guessed");
            System.out.println(inputPair);
            System.out.println("The secret numbers are");
            System.out.println(secretPair);
        }
   }
}
```

Automatic boxing allows you to use an `int` *argument for an* `Integer` *parameter.*

Multiple Type Parameters

- A generic class definition can have any number of type parameters.
  - Multiple type parameters are listed in angular brackets just as in the single type parameter case, but are separated by commas.

Multiple Type Parameters (Cont'd)

Display 14.8 Multiple Type Parameters

```
public class TwoTypePair<T1, T2>
{
    private T1 first;
    private T2 second;

    public TwoTypePair()
    {
        first = null;
        second = null;
    }

    public TwoTypePair(T1 firstItem, T2 secondItem)
    {
        first = firstItem;
        second = secondItem;
    }
    public void setFirst(T1 newFirst)
    {
        first = newFirst;
    }

    public void setSecond(T2 newSecond)
    {
        second = newSecond;
    }

    public T1 getFirst()
    {
        return first;
    }
```

(continued)

Multiple Type Parameters (Cont'd)

Display 14.8 **Multiple Type Parameters**

```
    public T2 getSecond()
    {
        return second;
    }

    public String toString()
    {
        return ( "first: " + first.toString() + "\n"
                + "second: " + second.toString() );
    }

    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            TwoTypePair<T1, T2> otherPair =
                        (TwoTypePair<T1, T2>)otherObject;
            return (first.equals(otherPair.first)
                && second.equals(otherPair.second));
        }
    }
}
```

The first **equals** *is the* **equals** *of the type* **T1**. *The second* **equals** *is the* **equals** *of the type* **T2**.

Bounds for Type Parameters

- Sometimes it makes sense to restrict the possible types that can be plugged in for a type parameter `T`.
  - For instance, to ensure that only classes that implement the **`Comparable`** interface are plugged in for **`T`**, define a class as follows:

```
public class RClass<T extends Comparable>
```

  - "**`extends Comparable`**" serves as a *bound* on the type parameter **`T`**.
  - Any attempt to plug in a type for **`T`** which does not implement the **`Comparable`** interface will result in a compiler error message.

Bounds for Type Parameters (Cont'd)

- A bound on a type may be a class name (rather than an interface name)
  - Then only descendent classes of the bounding class may be plugged in for the type parameters:

```
public class ExClass<T extends Class1>
```

- A bounds expression may contain multiple interfaces and up to one class.
- If there is more than one type parameter, the syntax is as follows:

```
public class Two<T1 extends Class1, T2 extends Class2 &
  Comparable>
```

Bounds for Type Parameters (Cont’d)

Display 14.10 A Bounded Type Parameter

```
public class Pair<T extends Comparable>
{
    private T first;
    private T second;

    public T max()
    {
        if (first.compareTo(second) <= 0)
            return first;
        else
            return second;
    }

  <All the constructors and methods given in Display 14.5
          are also included as part of this generic class definition>

}
```

Safe because T guarantees to implement comparable.

Generic Methods

- When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class.

- In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class

  – A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter.

  – The type parameter of a generic method is local to that method, not to the class.

Generic Methods (Cont'd)

- The type parameter must be placed (in angular brackets) after all the modifiers, and before the returned type:

```
public <T> T genMethod(T a)
```

- When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angular brackets

```
String s = NonG.<String>genMethod(c);
```

A Derived Generic Class: An Example

Display 14.11 A Derived Generic Class

```
public class UnorderedPair<T> extends Pair<T>
{
    public UnorderedPair()
    {
        setFirst(null);
        setSecond(null);
    }

    public UnorderedPair(T firstItem, T secondItem)
    {
        setFirst(firstItem);
        setSecond(secondItem);
    }
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            UnorderedPair<T> otherPair =
                            (UnorderedPair<T>)otherObject;
            return (getFirst().equals(otherPair.getFirst())
               && getSecond().equals(otherPair.getSecond()))
               ||
                  (getFirst().equals(otherPair.getSecond())
               && getSecond().equals(otherPair.getFirst()));
        }
    }
}
```

A Derived Generic Class: An Example (Cont'd)

Display 14.12 Using UnorderedPair

```
public class UnorderedPairDemo
{
    public static void main(String[] args)
    {
        UnorderedPair<String> p1 =
             new UnorderedPair<String>("peanuts", "beer");
        UnorderedPair<String> p2 =
             new UnorderedPair<String>("beer", "peanuts");
        if (p1.equals(p2))
        {
            System.out.println(p1.getFirst() + " and " +
                       p1.getSecond() + " is the same as");
            System.out.println(p2.getFirst() + " and "
                               + p2.getSecond());
        }
    }
}
```

Wildcards

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
    }
}
```

We want a method that prints our all the elements of a collection

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

Here is a naïve attempt at writing it using generics

```
printCollection(stones);
```

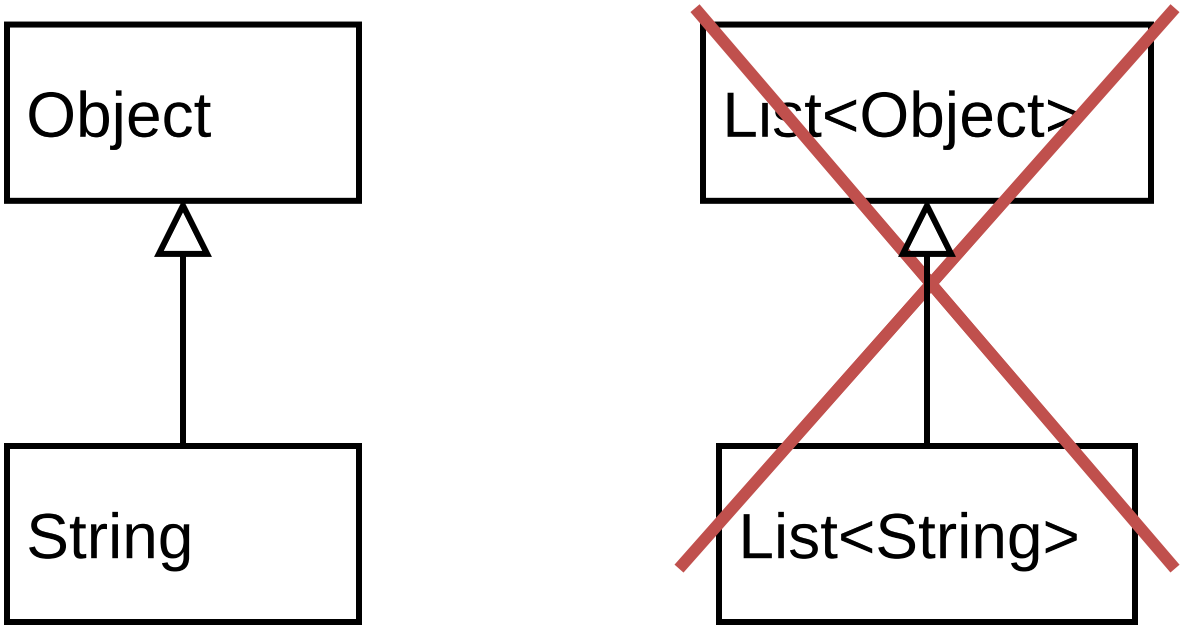
Won’t compile!

REMEMBER!!!!!!

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

List<Object> lo = ls;
```

Compile error as it is not type safe!

In other words…



So how do we do this??????????

What type matches all kinds of collections?

Collection<?>

"collection of unknown" is a collection whose element type matches anything — **a wildcard type**

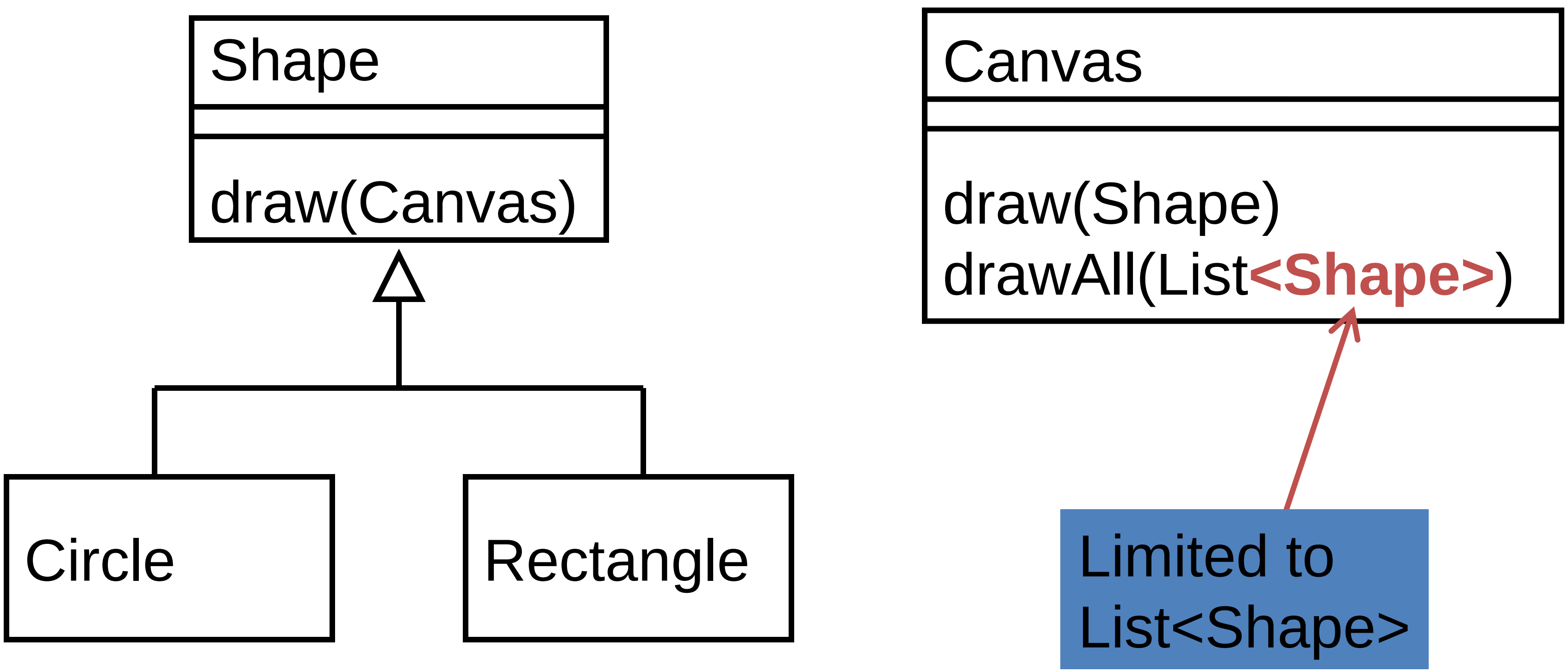
```
void printCollection(Collection<?> c) {
    for (Object e: c){
        System.out.println(e);
    }
}
```

```
printCollection(stones);
```

```
stone(java.awt.Color[r=255,g=0,b=0])
stone(java.awt.Color[r=0,g=255,b=0])
stone(java.awt.Color[r=0,g=255,b=0])
```

Bounded Wildcards

Consider a simple drawing application to draw shapes (circles, rectangles,…)



A Method that accepts a List of any kind of Shape…

```
public void drawAll(List<? extends Shape>) {…}
```

a bounded wildcard

Shape is the ***upper bound*** of the wildcard

More fun with generics

```
import java.util.*;
 …

    public void pushAll(Collection<? extends E> collection) {
        for (E element : collection) {
            this.push(element);
        }
    }

    public List<E> sort(Comparator<? super E> comp) {
        List<E> list = this.asList();
        Collections.sort(list, comp);
        return list;
    }
```

All elements must be *at least* an E

The comparison method must require *at most* an E

Generics

[How does it work? – "Erasure"]

There is no real copy for each parameterized type

What is being done?

- **<u>Compile time check</u> (e.g. List<Integer> adds only Integers)**
- **<u>Compiler adds</u> run-time casting (e.g. pulling item from List<Integer> goes through run-time casting to Integer)**
- **At run-time, the parameterized types (e.g. <T>) are <u>Erased</u> – this technique is called <u>Erasure</u>**

At run-time, List<Integer> is just a List !