Java Generics
Some more Examples

What are Generics?

- Generics abstract over Types
- Classes, Interfaces and Methods can be Parameterized by Types
- Generics provide increased readability and type safety (robustness)

Java Generic Programming

- Java has class Object
  - Supertype of all object types
  - This allows “subtype polymorphism”
    - Can apply operation on class T to any subclass S <: T
- Java 1.0 – 1.4 do not have templates
  - No parametric polymorphism
  - Many consider this the biggest deficiency of Java
- Java type system does not let you cheat
  - Can cast from supertype to subtype
  - Cast is checked at run time

list of lists with and without generics

List of lists without generics

```
List ys = new LinkedList();
ys.add("zero");
List yss;
yss = new LinkedList();
yss.add(ys);
String y = (String)
((List)yss.iterator().next()).iterator().next();

// Evil run-time error
Integer z = (Integer)ys.iterator().next();
```

We will realize it only at runtime

list of lists with generics

```
List<String> ys = new LinkedList<String>();
ys.add("zero");
List<List<String>> yss;
yss = new LinkedList<List<String>>();
yss.add(ys);
String y = yss.iterator().next().iterator().next();

// Compile-time error – much better!
Integer z = ys.iterator().next();
```

So we can avoid run time errors

generics also offer convenience

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

versus

```
List myIntList = new LinkedList ();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

We don’t have to do this casting

list implementation with and without generics

without generics

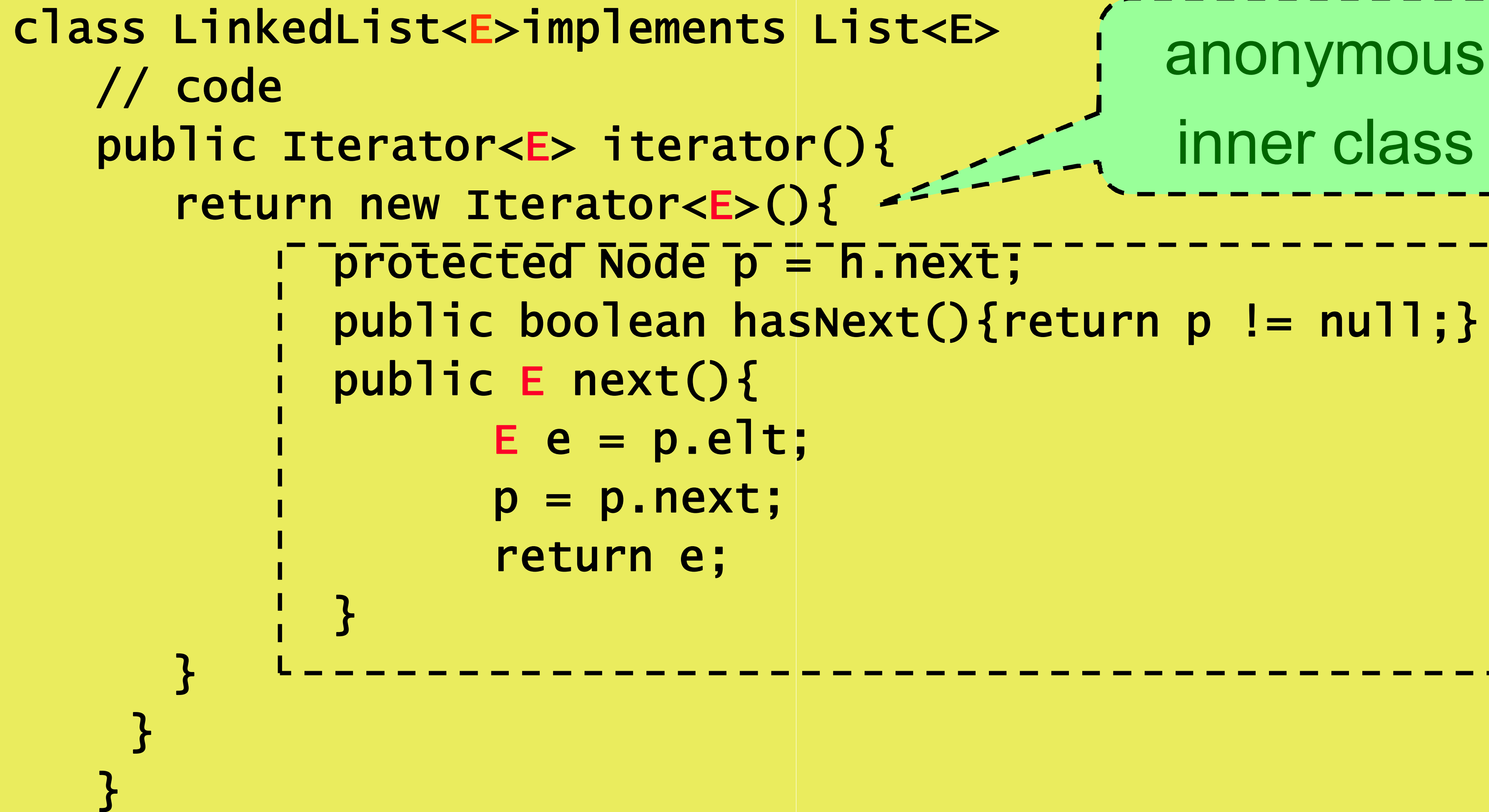
```
class LinkedList implements List {
  protected class Node {
            Object elt;
            Node next;
            Node(Object elt){elt = e; next = null;}
  }
  protected Node h, t;
  public LinkedList() {h = new Node(null); t = h;}
  public void add(Object elt){
      t.next = new Node(elt);
      t = t.next;
  }
}
```

note: see the inner class

with generics

```
class LinkedList<E>implements List<E>
  protected class Node {
      E elt;
      Node next;
      Node(E elt){elt = e; next = null;}
  }
  protected Node h, t;
  public LinkedList() {h = new Node(null); t = h;}
  public void add(E elt){
      t.next = new Node(elt);
      t = t.next;
  }
  // …
}
```

implementing the iterator with an anonymous inner class



Generics and Subtyping

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

is this correct?

- No! The second line will cause a compile time error
- Why? Consider the following scenario:

```
lo.add(new Object());
String s = ls.get(0); //attempts to assign an object to a String!
```

- We should not forget that collections change!

Bounded Parameterized Types

- The syntax MathBox<E extends Number> means that the type parameter of MathBox must be a subclass of the Number class
  - We say that the type parameter is bounded

```
new MathBox<Integer>(5); //Legal
new MathBox<Double>(32.1); //Legal
new MathBox<String>("No good!");//Illegal
```

Bounded Parameterized Types

- Inside a parameterized class, the type parameter serves as a valid type. So the following is valid.

```
public class OuterClass<T> {

  private class InnerClass<E extends T> {
    …
  }
  …
}
```

Syntax note: The <A extends B> syntax is valid even if B is an interface.

Bounded Parameterized Types

- Java allows multiple inheritance in the form of implementing multiple interfaces, so multiple bounds may be necessary to specify a type parameter. The following syntax is used then:

  <T extends A & B & C & …>

- Example

```
interface A {…}
interface B {…}

class MultiBounds<T extends A & B> {
…
}
```

Subclassing a generic class

```
import java.awt.Color;

public class Subclass extends MyClass<Color> {

    // You almost always need to supply a constructor
    public Subclass(Color color) {
        super(color);
    }

    public static void main(String[ ] args) {
        Subclass sc = new Subclass(Color.GREEN);
        sc.print(Color.WHITE);
    }
}
```

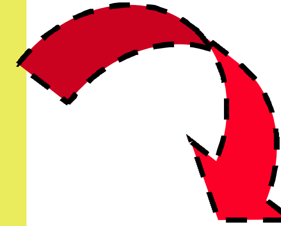
Wildcards

- Consider the problem of writing code that prints out all the elements in a collection

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  for (k = 0; k < c.size(); k++) {
      System.out.println(i.next());
  }
}
```

1st Naïve Try with Generics

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  for (k = 0; k < c.size(); k++) {
      System.out.println(i.next());
  }
}
```

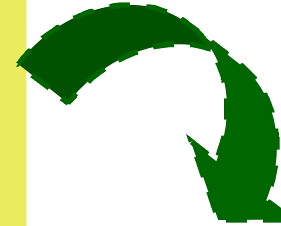


```
void printCollection(Collection<Object> c)
  {
  for (Object e : c) {
      System.out.println(e);
  }
}
```

- The problem is that this new version is much less useful than the old one.
- The old code could be called with any kind of collection as a parameter,
- The new code only takes Collection<Object>, which, is not a supertypeof all kinds of collections!

Correct way – Use Wildcards

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  for (k = 0; k < c.size(); k++) {
      System.out.println(i.next());
  }
}
```



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

- So what is the supertype of all kinds of collections?
- Pronounced “collection of unknown” and denoted Collection<?>,
- A collection whose element type matches anything.
- It’s called a wildcard type for obvious reasons.

Using Wildcards Again

```
class Person { ...}
class Driver extends Person { ...}

public class Census {
public static void addRegistry(Map<String, ? extends Person> registry) { ...}
}...

Map<String, Driver> allDrivers = ...;

Census.addRegistry(allDrivers);
```

however, ...

```
Collection <?> c = new ArrayList<String>();
c.add(new Object()); // compile time error
```

- As we don't know what the element type of c stands for, we cannot add objects to it
- On the other hand, given a `List<?>` we can call **`get()`** and make use the result. Although the result type is an unknown type, we are sure that it is an object.

analogous case

```
public void addRectangle(List <? extends Shape> shapes){
  shapes.add(0, new Rectangle()); // compile-time error
}
```

- As we don't know what the element type of list stands for, we cannot add a Rectangle (we cannot be sure that the parameter type will be a supertype of Rectangle).

Generic methods

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

- We can call this method with any kind of collection whose element type is a supertype of the element type of the array.
- We dont have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the types of the actual parameters (it will infer the most specific type that will make the call type-correct)

(cont)

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T inferred to be Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs);// T inferred to be String
fromArrayToCollection(sa, co);// T inferred to be Object
Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
fromArrayToCollection(ia, cn);// T inferred to be Number
fromArrayToCollection(fa, cn);// T inferred to be Number
fromArrayToCollection(na, cn);// T inferred to be Number
fromArrayToCollection(na, co);// T inferred to be Object
fromArrayToCollection(na, cs);// compile-time error
```

Implementing Generics

- Type erasure
  - Compile-time type checking uses generics
  - Compiler eliminates generics by erasing them
    - Compile List<T> to List, T to Object, insert casts
- “Generics are not templates”
  - Generic declarations are typechecked
  - Generics are compiled once and for all
- How do generics affect my code?
  - They don’t – except for the way you’ll code!
  - Non-generic code can use generic libraries;
  - For example, existing code will run unchanged with generic Collection library

example of type erasure

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

- will print `true`!
- Why ?
- Erasure erases all generics type argument information at compilation phase.
  - E.g. List<String> is converted to List
  - E.g. String t = stringlist.iterator().next() is converted to String t = (String) stringlist.iterator().next()
- As part of its translation, a compiler will map every parameterized type to its type erasure

For more see

- http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf