Generics

Sang Shin
JPassion.com
“Code with Passion!”

Topics

- What is and why use Generics?
- Usage of Generics
- Creating your own generic class
- Creating your own generic method
- Generics and sub-typing
- Wildcard
- Bounded wildcard – upper bounded, lower bounded
- Type erasure
- Interoperability with non-generic code

What is it?
How do define it?
How to use it?
Why use it?

What is Generics?

- Generics provides parameterization of Types
  - Classes, Interfaces and Methods can be Parameterized by Types
- Similar to the way a type itself is parameterized by an instance of it, for example
  - *public void myMethod (String x); // Definition of a method*
  - *myMethod ("hello");)   // Usage of a method*
- Generics allows type parameterization
  - *public void myMethod (T t);   // Generic type*
  - *myMethod (new Integer(1));   // Specific type*
  - *myMethod (new String("hello")); // Specific type*

What is Generics?

- Generics makes type safe code possible
  - If it compiles without any errors or warnings, then it must not raise any unexpected ClassCastException during runtime
- Generics provides increased readability on types
  - It shows type information

Definition of a Generic Class: LinkedList<E>

- Definition: LinkedList<E> has a type parameter E that represents the type of the elements stored in the linked list – in other words, E represents a type not an instance

```
public class LinkedList<E>
   extends AbstractSequentialList<E>
   implements List<E>, Queue<E>, Cloneable, java.io.Serializable {

   private transient Entry<E> header = new Entry<E>(null, null, null);
   private transient int size = 0;

   public E getFirst() {
       if (size==0) throw new NoSuchElementException();
       return header.next.element;
   }
  ...
```

Usage of Generic Class: LinkedList<Integer>

- Usage: Replace type parameter <E> with concrete type argument, like <Integer> or <String> or <MyType>
  - LinkedList<Integer> can store only Integer or sub-type of Integer as elements

```
LinkedList<Integer> li =
              new LinkedList<Integer>();
li.add(new Integer(0));
Integer i = li.iterator().next();
```

Example: Definition and Usage of Parameterized *List* interface

```
// Definition of the Generic'ized
// List interface
//
interface List<E>{
   void add(E x);
  Iterator<E> iterator();
   ...
}
// Usage of List interface with
// concrete type parameter,String
//
List<String> ls = new ArrayList<String>(10);
```

Type parameter

This is type parameterization

Type argument

Why Generics? Non-genericized code could cause a crash during runtime

```
// Suppose you want to maintain only String
// entries in a Vector.  By mistake,
// you add an Integer element.  Compiler
// does not detect this and could cause
// ClassCastException during runtime.

Vector v = new Vector();
v.add(new String("valid string")); // intended
v.add(new Integer(4));             // unintended

// ClassCastException occurs during runtime
String s = (String)v.get(1);
```

Why Generics?

- Problem (in non-Genericized code): Collection element types
  - Compiler is unable to verify types of the elements
  - Assignment must have type casting
  - *ClassCastException* can occur during runtime

- Solution: Generics
  - Tell the compiler the type of the collection elements
  - Let the compiler do the casting
  - Example: Compiler will check if you are adding Integer type entry to a String type collection
    - Compile time detection of type mismatch is now possible

Usage of Generics

Using Generic Classes: Example 1

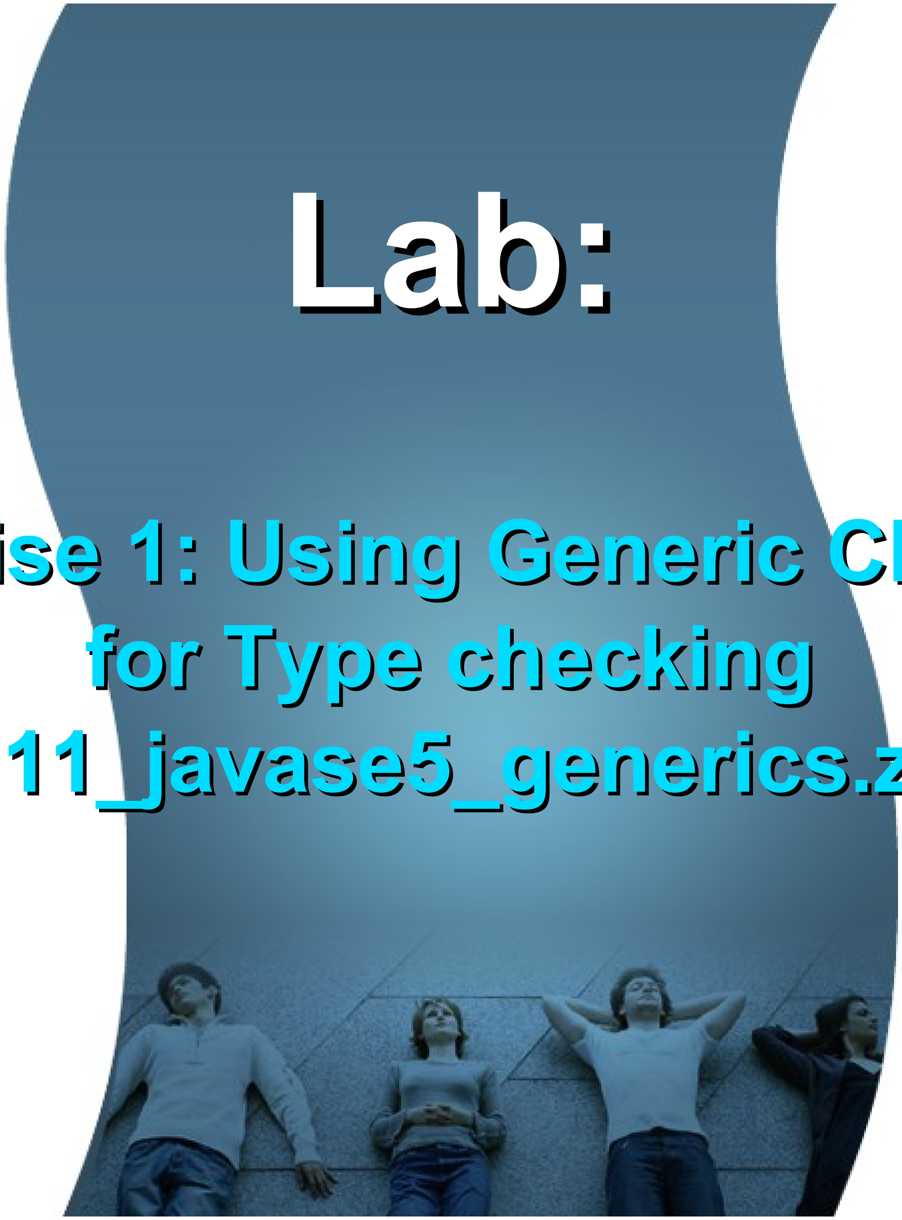
- Instantiate a generic class to create type specific object
- In Java SE 5+, all collection classes are rewritten to be generics-aware classes

```
// Create ArrayList of String type
List<String> ls = new ArrayList<String>();
ls.add(new Integer(5)); // Error detected at
                        // compile time
ls.add(new String(“hello”));
String s = ls.get(0);   // No casting needed
```

Using Generic Classes: Example 2

- Generic class can have multiple type parameters
- Type argument can be a custom type, like Mammal class in the example below

```
// Create HashMap with two type parameters
HashMap<String, Mammal> map =
          new HashMap<String, Mammal>();
map.put("wombat", new Mammal("wombat"));

Mammal w = map.get(“wombat”);
```



Lab:

Exercise 1: Using Generic Classes for Type checking
1111_javase5_generics.zip

Creating Your Own Generic Class Part I

So I want to create Generic'ized class..

- Generic programming produces highly general and reusable code
  - Like generic'ized Collection classes are highly general and reusable (compared to non-generic'ized version)
- Examples
  - Want to write “sorting” method, which can sort items of any type
  - Want to write “display” method, which can print an object of any type

Without using Generics

- Without using Generics, you have to write custom classes for each type if you want type checking at compile time

```
public class SomethingInteger {
    private Integer thing;
    public void setThing(Integer thing) {
        this.thing = thing;
    }
}

class SomethingString {
    private String thing;
    public void setThing(String thing) {
        this.thing = thing;
    }
}
```

Definition of classes

```
// Create object from non-genericized class
SomethingInteger s1 = new SomethingInteger();
s1.setThing(new Integer(3));
// s1.setThing(new String("test")); // compile error

// Create object from non-genericized class
SomethingString s2 = new SomethingString();
// s2.setThing(new Integer(3)); // compile error
s2.setThing(new String("test"));
```

Usage of classes

With Generics

- With Generics, you create one generic class - then parameterize the type

```
public class SomethingGeneric <T> {
    private T thing;
    public void setThing(T thing) {
        this.thing = thing;
    }
}
```

- Then, provide the type argument

```
SomethingGeneric<Integer> s3 = new SomethingGeneric<Integer>();
s3.setThing(new Integer(3));
// s3.setThing(new String("test")); // compile error

SomethingGeneric<String> s4 = new SomethingGeneric<String>();
// s4.setThing(new Integer(3)); // compile error
s4.setThing(new String("test"));
```

So I want to write type-safe code

- Now I want to write type safe code with a different type, Long
- Without Generics, you have to create one more custom class

```
public class SomethingLong {
    private Long thing;
    public void setThing(Long thing) {
        this.thing = thing;
    }
}
```

- With Generics, just use <Long> type argument

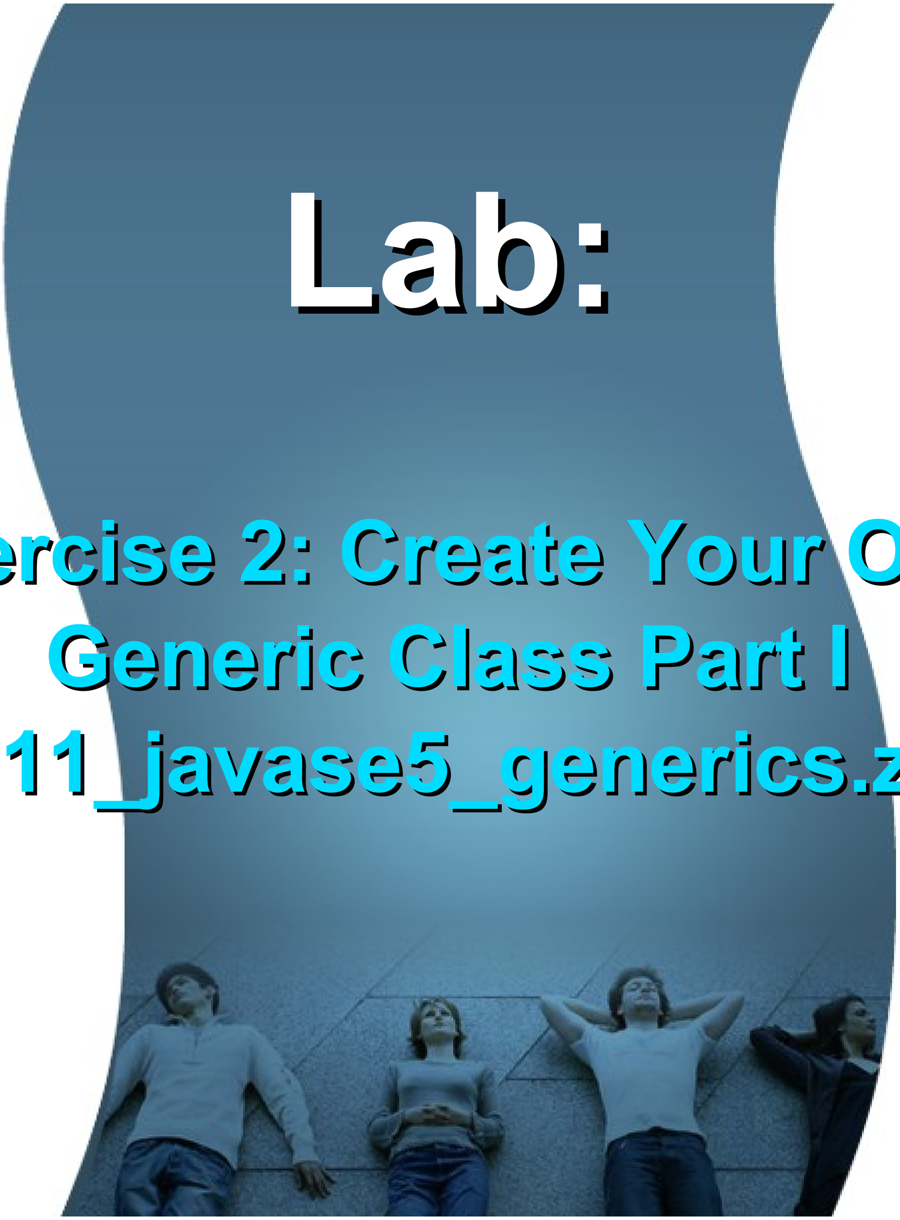
```
SomethingGeneric<Long> s3 = new SomethingGeneric<Long>();
s3.setThing(new Long(3L));
// s3.setThing(new String("test")); // compile error
```

Type Parameter Naming Convention

- Single uppercase letters for type parameters
  - Otherwise, it would look like a class or interface name
- Recommended conventions
  - E for element (used in Java Collections Framework)
  - K for key
  - N for number
  - T for type
  - V for value
  - S, U, V etc – 2nd, 3rd, 4th types

Lab:

Exercise 2: Create Your Own Generic Class Part I
1111_javase5_generics.zip



Type Inference

Type Inference

- From Java 7, type inference can be performed by compiler
- Without type inference

```
List<Person> people = new ArrayList<Person>();
people.add(new Person(“sang”,100));
people.add(new Perrson(“jon”, 34);
```

- With type inference

```
List<Person> people = new ArrayList<>();
people.add(new Person(“sang”,100));
people.add(new Perrson(“jon”, 34);
```

Creating Your Own Generic Class Part II

Defining Your Own Generic Class

```
public class Pair<F, S> {
   F first;  S second;

   public Pair(F f, S s) {
      first = f;  second = s;
   }

   public void setFirst(F f){
      first = f;
   }

   public F getFirst(){
      return first;
   }

   public void setSecond(S s){
      second = s;
   }

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

Class definition
Types are defined with < >.

This class uses two
type parameters internally

Parameters and return
types are without < >

Using Your Own Generic Class – You Provide concrete type arguments

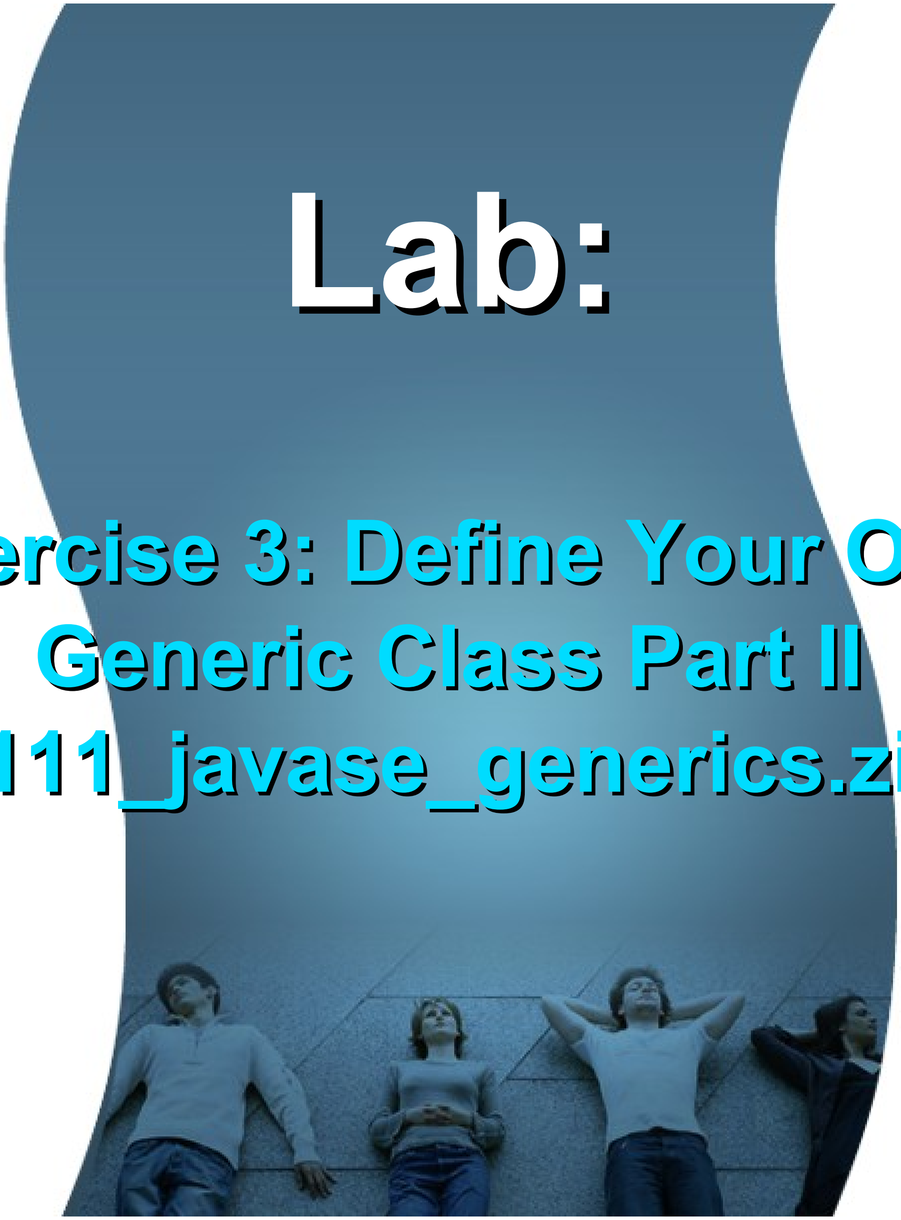
```
public class Main {

    public static void main(String[] args) {

        // Create an instance of Pair <F, S> class.  Let's call it p1.
        Number n1 = new Integer(5);
        String s1 = new String("Sun");
        Pair<Number,String> p1 = new Pair<>(n1, s1);
        // Pair<Number,String> p2
        //    = new Pair<>(new Integer(4), new Integer(3)); // compile error

        // Set internal variables of p1.
        p1.setFirst(new Long(6L));
        p1.setSecond(new String("rises"));
        //p1.setFirst(new String("error"));    // Compile error expected
        //p1.setSecond(new Long(5L));      // Compile error expected
    }
}
```

Extending a Generic Class

```
public class PairExtended <F, S, T> extends Pair<F, S> {

    T third;

    /** Creates a new instance of PairExtended */
    PairExtended(F f, S s, T t){
        super(f, s);
        third = t;
    }

    public T getThird(){
        return third;
    }
}
```

Use a Generic Class as a Type Argument

```
// Create an instance of PairExtended<F. S, T> class with
// with ArrayList<E> as a third type argument.
ArrayList<Integer> ar4 = new ArrayList<>();
ar4.add(6000);
ar4.add(7000);
PairExtended<Number, String, ArrayList<Integer>> pe5
            = new PairExtended<Number, String, ArrayList<Integer>>(n4, s4, ar4);
```



Lab:

Exercise 3: Define Your Own Generic Class Part II
1111_javase_generics.zip

Creating Your Own Generic Method

Creating Generic Method

- Useful when you have a non-generic class and you want to generic'ize a particular method of that class
- Generic methods are similar to generic classes
  - They are different only in one aspect that scope of type information is inside method only
- Generic methods are methods that introduce their own type parameters

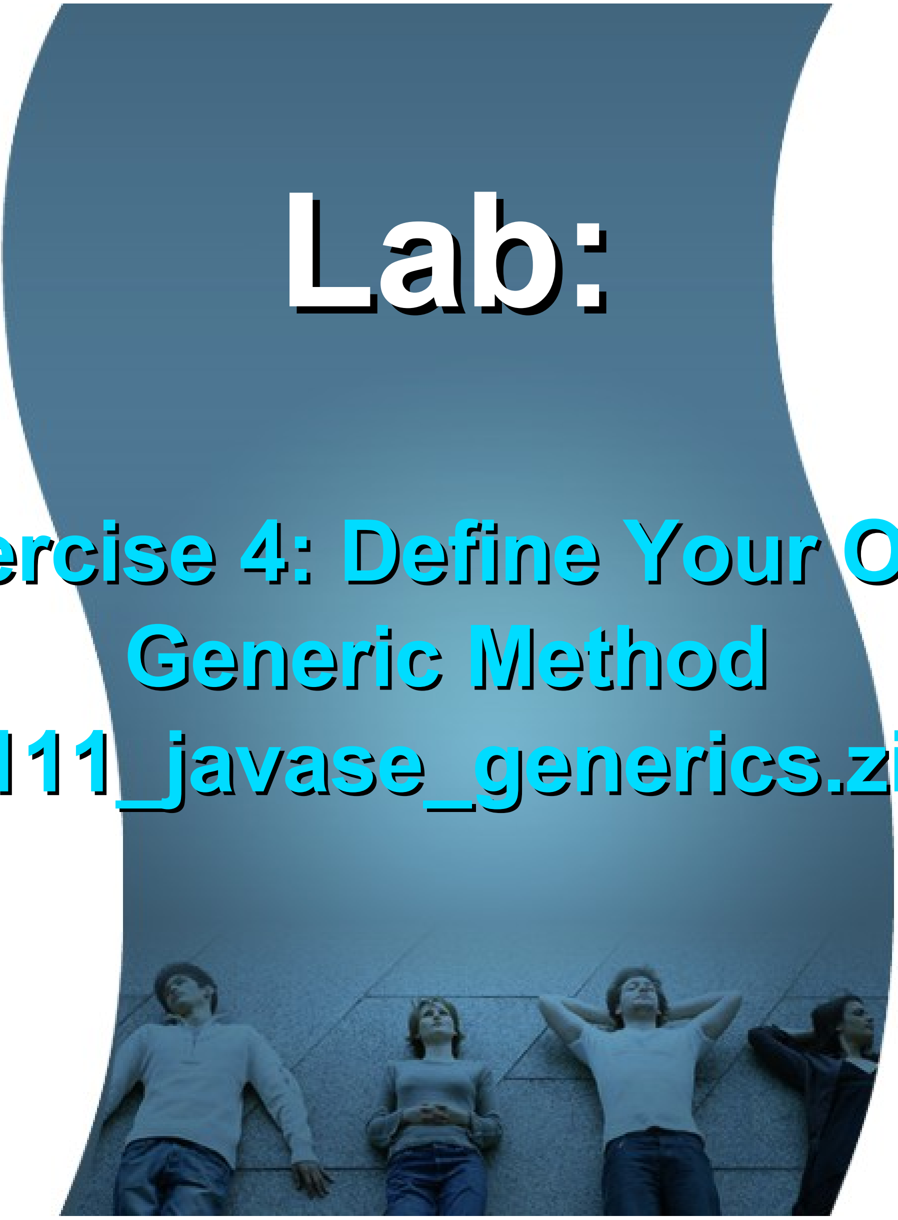
```
public static <T> int countAllOccurrences(T[ ] list, T item) {
  int count = 0;
  for (T listItem : list){
    if (item.equals(listItem)) {
      count++;
    }
  }
  return count;
}
```

This indicates that the T is a type parameter. Without it, compiler will consider T as regular type.

Lab:

Exercise 4: Define Your Own Generic Method

1111_javase_generics.zip



Sub-typing

Generics and Sub-typing

- You can do this (using pre-J2SE 5.0 Java)
  - Object o = new Integer(5);
- You can even do this (using pre-J2SE 5.0 Java)
  - Object[] or = new Integer[5];
- So you would expect to be able to do this (Well, you can't do this!!!)
  - ArrayList<Object> ao = new ArrayList<Integer>();
  - This is counter-intuitive at the first glance

Generics and Sub-typing

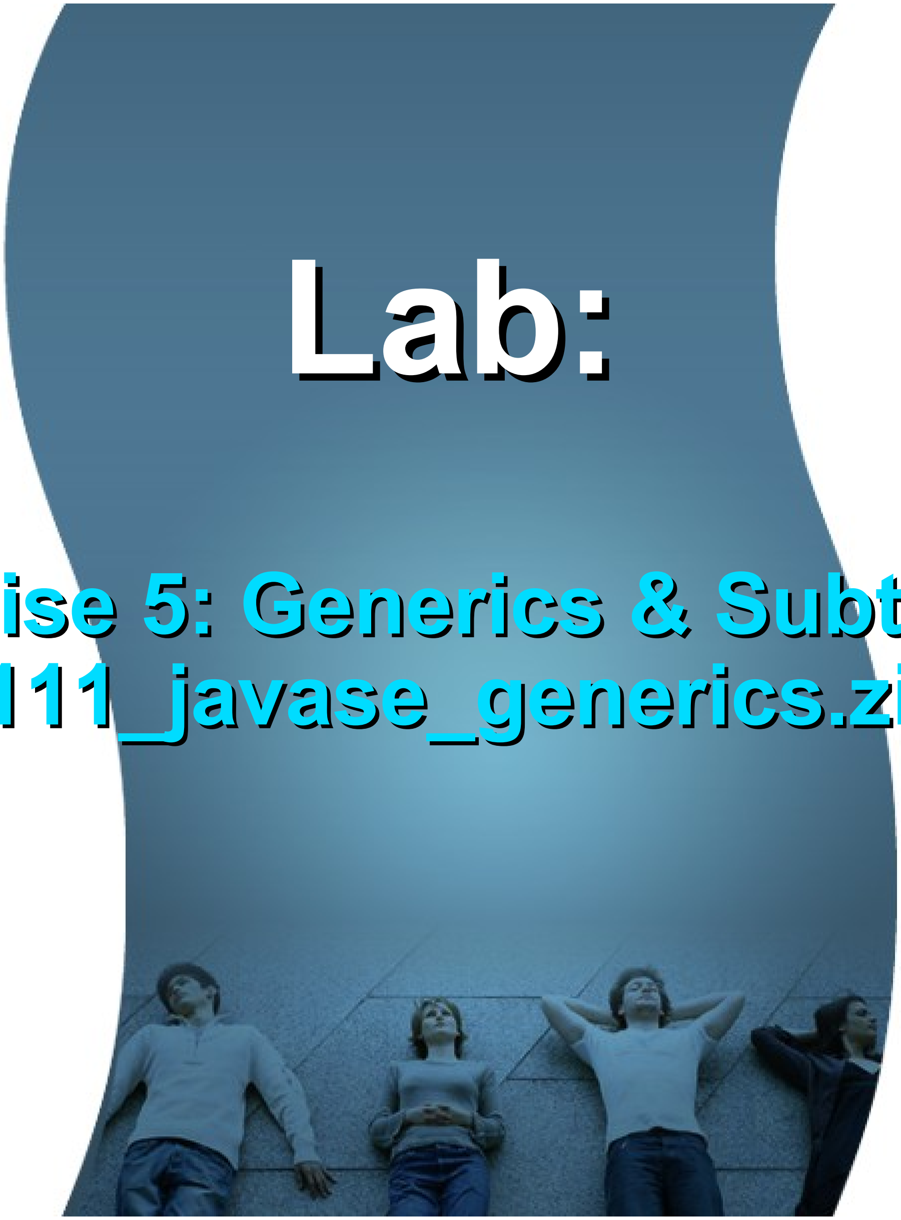
- Why this compile error? It is because if it is allowed, *ClassCastException* can occur during runtime – this is not type-safe
  > ArrayList<Integer> ai = new ArrayList<Integer>();
  > ArrayList<Object> ao = ai; // If ArrayList<Object> ao = new ArrayList<Integer>();
  >   // is allowed, this should be allowed
  > ao.add(new Object());   // Now an instance of Object class
  > Integer i = ai.get(0); // This would result in
  >   // runtime ClassCastException
- So there is no inheritance relationship between type arguments of a generic class

Generics and Sub-typing

- The following code work
  - ArrayList<Integer> ai = new ArrayList<Integer>();
  - List<Integer> li2 = new ArrayList<Integer>();
  - Collection<Integer> ci = new ArrayList<Integer>();
  - Collection<String> cs = new Vector<String>(4);
- Inheritance relationship between classes themselves still exists
  - No change in this, hence there is absolutely no backward/forward compatibility issue with Generics

Generics and Sub-typing

- The following code work
  - ArrayList<Number> an = new ArrayList<Number>();
  - an.add(new Integer(5)); // OK
  - an.add(new Long(1000L)); // OK
  - an.add(new String(“hello”)); // compile error
- Entries in a collection still maintain inheritance relationship
  - No change in this, hence there is absolutely no backward/forward compatibility issue with Generics



Lab:

Exercise 5: Generics & Subtyping
1111_javase_generics.zip

Wildcard

Why Wildcards? Problem

- Consider the problem of writing a method that prints out all the elements in a collection
- Here's how you might write it in an older version of the language (i.e., a pre-5.0 release):

```
// pre-5.0 code
static void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```

Why Wildcards? Let's see a Problem

- And here is a naive attempt at writing it using generics
  Well.. You can't do this!

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

public static void main(String[] args) {

  Collection<String> cs = new ArrayList<String>();
  printCollection(cs); // Compile error because
  // Collection<Object> = new ArrayList<String>() is not allowed

  List<Integer> li = new ArrayList<Integer>(10);
  printCollection(li); // Compile error because
  // Collection<Object> = new ArrayList<Integer>(10) is not allowed
}
```

Why Wildcards? Solution

- Use Wildcard type argument <?>
- Collection<?> means Collection of unknown type
- Accessing entries of Collection of unknown type with Object type is safe

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

public static void main(String[] args) {

  Collection<String> cs = new ArrayList<String>();
  printCollection(cs); // No Compile error

  List<Integer> li = new ArrayList<Integer>(10);
  printCollection(li); // No Compile error
}
```

More on Wildcards

- You cannot access entries of Collection of unknown type other than Object type

```
static void printCollection(Collection<?> c) {
  for (String o : c) // Compile error
    System.out.println(o);
}

public static void main(String[] args) {

  Collection<String> cs = new ArrayList<String>();
  printCollection(cs); // No Compile error

  List<Integer> li = new ArrayList<Integer>(10);
  printCollection(li); // No Compile error
}
```

Bounded Wildcard

Two Types of Bounded Wildcards

- Upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the extends keyword – it is called upper bounded because the specified class (Number class below) is the top of the class hierarchy

```
List<? extends Number> foo = new ArrayList<Number>();
List<? extends Number> foo = new ArrayList<Integer>();
List<? extends Number> foo = new ArrayList<Double>();
```

- Lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type – it is called lower bounded because the specified class (Number class below) is the bottom of the class hierarchy

```
List<? super Number> foo = new ArrayList<Number>();
List<? super Number> foo = new ArrayList<Object>();
```

Upper bounded wildcard

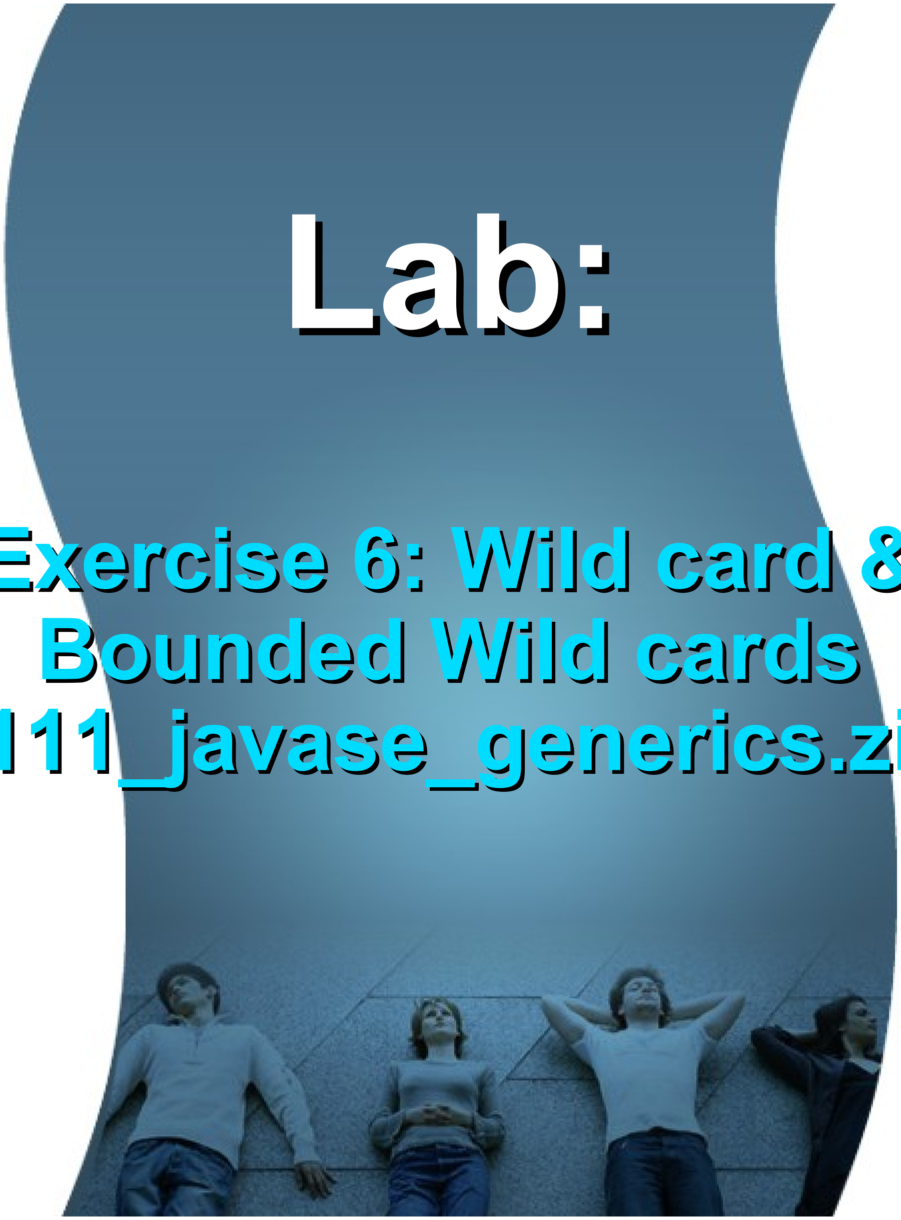
- You can read but you cannot write - You can't add any object to List<? extends T> because you can't guarantee what kind of List it is really pointing to, so you can't guarantee that the object is allowed in that List. The only "guarantee" is that you can only read from it and you'll get a T or subclass of T

```
// So foo can point to any of following three types

List<? extends Number> foo = new ArrayList<Number>();
List<? extends Number> foo = new ArrayList<Integer>();
List<? extends Number> foo = new ArrayList<Double>();

// You can't add an Integer to foo because foo could be pointing at a List<Double>
foo.add(new Integer(1)); // compile error
// You can't add an Double to foo because foo could be pointing at a List<Integer>
foo.add(new Double(1.0)); //compile error

// But you can read
foo.get(1);
```

Lower bounded wildcard

- You can write but you cannot read - You can't read the specific type T (e.g. Number) from List<? super T> because you can't guarantee what kind of List it is really pointing to. The only "guarantee" you have is you are able to add a value of type T (or subclass of T) without violating the integrity of the list being pointed to.

```
List<? super Number> foo = new ArrayList<Number>();
List<? super Number> foo = new ArrayList<Object>();

// You can write
foo.add(new Integer(1));
foo.add(new Double(1.0));

// But you cannot read it into a specific type
Number number = foo.get(1);
// But you can read it into Object type though
Object object = foo.get(1);
```



Lab:

Exercise 6: Wild card & Bounded Wild cards 1111_javase_generics.zip

Raw Type & Type Erasure

Raw Type for backward compatibility

- Generic type instantiated with no type arguments
- Pre-J2SE 5.0 classes continue to function as raw type
- Compiler generates a warning message, however

```
// Generic type instantiated with type argument
List<String> ls = new LinkedList<String>();

// Generic type instantiated with no type
// argument. This is Raw type
List lraw = new LinkedList();
```

Type Erasure

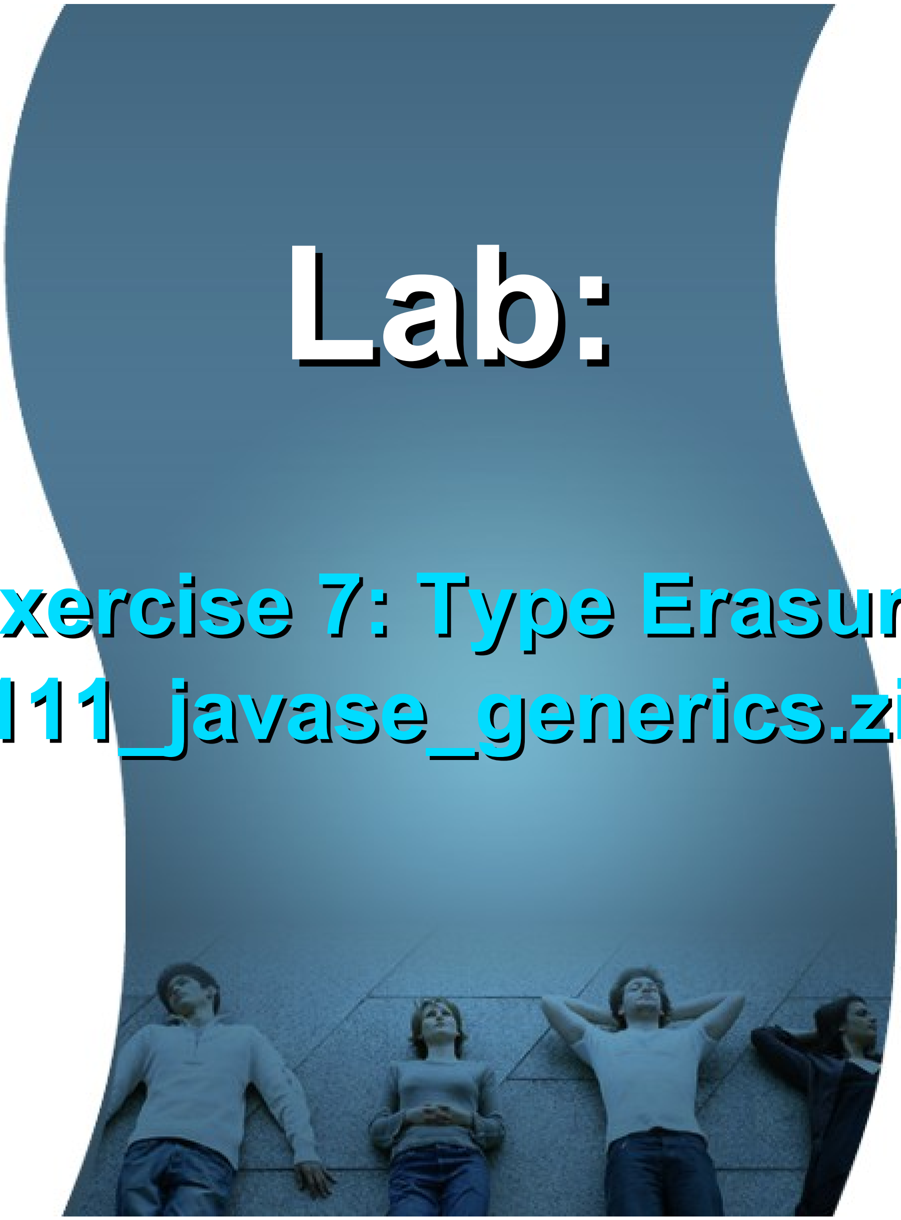
- All generic type information is removed in the resulting byte-code after compilation
- So generic type information does not exist during runtime
- After compilation, they all share same class
  - A class bytecode that represents ArrayList<String>, ArrayList<Integer> is the same class that represents ArrayList

Type Erasure Example Code: True or False?

```
ArrayList<Integer> ai = new ArrayList<Integer>();
ArrayList<String> as = new ArrayList<String>();
// Compare class object of ai and class object of as
Boolean b1 = (ai.getClass() == as.getClass());
System.out.println("Do ArrayList<Integer> and ArrayList<String> share
    same class? " + b1);
```

Type-safe Code Again

- The compiler guarantees that either:
  - The code it generates will be type-correct at run time, or
  - It will output a warning (using Raw type) at compile time
- If your code compiles without warnings and has no casts, then you will never get a ClassCastException during runtime
  - This is “type safe” code



Lab:

Exercise 7: Type Erasure
1111_javase_generics.zip

Interoperability with non-generic code

What Happens to the following Code?

```
import java.util.LinkedList;
import java.util.List;

public class GenericsInteroperability {

    public static void main(String[] args) {

        List<String> ls = new LinkedList<String>();
        List lraw = ls;
        lraw.add(new Integer(4));
        String s = ls.iterator().next();
    }

}
```

Compilation and Running

- Compilation results in a warning message
  - > *GenericsInteroperability.java* uses unchecked or unsafe operations.
- Running the code
  - > *ClassCastException*

Lab:

Exercise 8: Interoperability
1111_javase_generics.zip

Code with Passion!
JPassion.com