Comp 401
Advanced Generics

Instructor: Prasun Dewan

Prerequisite

- Generics

Java Generic Types (Review)

A scope name (class, interface, method) can be succeeded by a series of type parameters within angle brackets <A, B, C, …> that have the same value in all places

```
public interface I<T> {
        public void addElement (T t);
        public T elementAt (int index);
        public int size();
    }
```

A method, class, interface with type parameter is called a generic.

Create an elaboration of generic by giving actual value to type parameter

A single implementation is shared by all of its elaborations.

```
I<String>  stringI;
I<Point> pointI;
```

Assigning values to type parameters is a compile time activity for type checking only.

AHistory

```
public class AHistory<T> implements History<T> {
    public final int MAX_SIZE = 50;
    Object[] contents = new Object[MAX_SIZE];
    int size = 0;
    public int size() {return size;}
    public T elementAt (int index) {
        return (T) contents[index];
    }
    boolean isFull() {return size == MAX_SIZE;}
    public void addElement(T element) {
      if (isFull())
        System.out.println("Adding item to a full history");
      else {
        contents[size] = element;
        size++;
    }
}
```

A single implementation where T = Object is shared by all of its elaborations.

Instantiating A String History

```
History<String>  stringHistory = new AHistory<String>();
History<Point>   pointHistory  = new AHistory<Point>();
```

Instantiating A String History vs. Array

```
History<String>  stringHistory = new AHistory();
History<Point>   pointHistory  = new AHistory();
```

```
String[] strings = new String[50];
double[] doubles = new double[50];
```

Different implementations of different sizes.

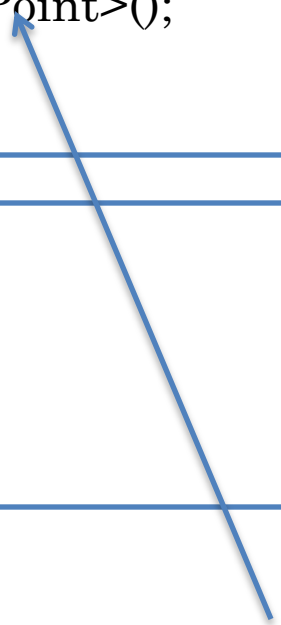
Pain to specify type each time

Since giving value to type parameter is compile time activity for type checking and only one kind of implementation with T = Object exists, do not have to (but can) specify value while instantiating.

Instantiating A String History vs. Array

```
History<String>  stringHistory = new AHistory<String>();
History<Point>   pointHistory  = new AHistory<Point>();
```

```
String[] strings = new String[50];
double[] doubles = new double[50];
```



Can specify value while instantiating

Extensions that Elaborate and Structural Type Equivalence

```
public interface StringHistory extends History<String> {}
```

```
public class AStringHistory extends AHistory<String> implements
StringHistory {}
```

```
StringHistory  stringHistory = new AStringHistory();
History<String>  history = new AHistory();
```

```
history = stringHistory;
stringHistory = history;
stringHistory = new History();
```

Extends != Equals

Java does not use structural equivalence

Integer and String Adders

```
public interface IntegerAdder {
 public Integer sum(Integer val1, Integer val2);
}
```

```
public interface StringAdder {
  public String sum(String val1, String val2);
}
```

```
public class AnIntegerAdder implements IntegerAdder{
  public Integer sum(Integer val1, Integer val2) {
    return val1 + val2;
  }
}
```

```
public class AStringAdder implements StringAdder{
  public String sum(String val1, String val2) {
    return val1 + val2;
  }
}
```

Integer and String Adders

```
public interface GenericAdder<OperandType> {
  public OperandType sum(OperandType val1,
                         OperandType val2);
}
```

```
public class AnIntegerAdder
                        implements GenericAdder<Integer>{
  public Integer sum(Integer val1, Integer val2) {
    return val1 + val2;
  }
}
```

```
public class AStringAdder
                       implements GenericAdder<String> {
  public String sum(String val1, String val2) {
    return val1 + val2;
  }
}
```

Integer and String Adders

```
public interface GenericAdder<OperandType> {
   public OperandType sum(OperandType val1,
                          OperandType val2);
}
```

```
public class AGenericAdder<OperandType> implements
GenericAdder<OperandType> {
public OperandType sum(OperandType val1, OperandType val2)
{
   return val1 + val2;
}
```

A single sum exists in AGenericAdder with OperandType = Object

+ not defined for String, Integer, but not Objects

An Integer and String Adder

```
public interface IntegerAdder {
 public Integer sum(Integer val1, Integer val2);
}
```

```
public interface StringAdder {
  public String sum(String val1, String val2);
}
```

```
public class AnIntegerAndStringAdder
      implements IntegerAdder, StringAdder {

  public Integer sum(Integer val1, Integer val2) {
    return val1 + val2;
  }
  public String sum(String val1, String val2) {
    return val1 + val2;
  }
}
```

An Integer and String Adder

```
public interface GenericAdder<OperandType> {
  public OperandType sum(OperandType val1,
                         OperandType val2);
}
```

```
public class AnIntegerAndStringAdder
     implements GenericAdder<Integer>, GenericAdder<String>{
  public Integer sum(Integer val1, Integer val2) {
    return val1 + val2;
  }
  public String sum(String val1, String val2) {
    return val1 + val2;
  }
}
```

A single sum exists in GenericAdder with OperandType = Object

Same method cannot be implemented twice (overloading rule)

Which method should the compiler choose?

Collection Type Rules

```
History<String>  stringHistory   = new AHistory();
History<Object> objectHistory = stringHistory;
```

```
stringHistory.addElement("hello");
objectHistory.addelement(new ACartesianPoint(5,10));
```

objectHistory and stringHistory refer to the same history;

This history now has a Point and a String!

T1 IS-A T2 does not imply collection of T1 IS-A T2

Type Rules

```
History<String>  stringHistory   = new AHistory();
History<Object> objectHistory = stringHistory;
```

```
stringHistory.addElement("hello");
objectHistory.addelement(new ACartesianPoint(5,10));
```

Problem occurs in some situations.

No problem in other situations.

Type Rules

```
void printX (History<Point> pointHistory) {
        for (int index = 0; index < pointHistory.size(); index++)
                System.out.println(pointHistory.elementAt(index).getX());
}
```

```
History<BoundedPoint> boundedPointHistory = new AHistory();
printX(boundedPointHistory);
```

Disallowed even though no type violations occur.

Can we type in such a way that safe operations on collections are allowed and unsafe are not?

Wild Cards

```
void printX (History<? extends Point> pointHistory) {
        for (int index = 0, index < pointHistory.size(); index++)
                System.out.println(pointHistory.elementAt(index).getX());
}
```

```
History<BoundedPoint> boundedPointHistory = new History();
printX(boundedPointHistory);
```

println() works on any type so does not matter what the unknown type of element is

A History whose elements are unknown subtypes of Shape.

Wild Cards

```
void addAll (History<? extends Point> pointHistory) {
        for (int index = 0, index < pointHistory.size(); index++)
                pointHistory.addElement(new ACartesianPoint(x,y));
}
```

```
History<BoundedPoint> boundedPointHistory = new History();
addAll(boundedPointHistory);
```

Not ACarestianPoint IS-A BoundedPoint

Add expects the argument to be of the same type as the type of the element, not any subtype of Point

A History whose elements are unknown subtypes of Point.

Collection Type Rules

```
History<String>  stringHistory   = new AHistory();
History<Object> objectHistory = stringHistory;
```

```
stringHistory.addElement("hello");
objectHistory.addelement(new ACartesianPoint(5,10));
```

objectHistory and stringHistory refer to the same history;

This history now has a Point and a String!

T1 IS-A T2 does not imply collection/array of T1 IS-A T2

Array Type Rules

```
String[] strings   = new String[50];
Object[] objects   = strings;

strings[0] = "hello";
objects[1] = new ACartesianPoint(5,10);
```

objects and strings refer to the same array.

This array now has a Point and a String!

ArrayStoreException

In the case of arrays, type checking done at run time maybe because wildcards don't work

Wild Cards

```
void printX (History<? extends Point> pointHistory) {
        for (int index = 0; index < pointHistory.size(); index++)
                System.out.println(pointHistory.elementAt(index).getX());
}
```

```
void printX (Point[] points) {
        for (int index = 0; index < points.length; index++)
                System.out.println(points[index].getX());
}
```

Array is not a generic so wildcards don't work

Runtime error can be given because different representation of different types of arrays.

Elaborating Instantiation

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

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

```
String  nonExistingElement = (new ArrayList()).get(0);
```

```
String  nonExistingElement = (new ArrayList<String>()).get(0);
```

IndexOutOfBoundsException

More Convincing Example

```
List<String> contents = new ArrayList();
contents.add("test");
List<String> correctCopy = new ArrayList(contents);
List<Point> incorrectCopy= new ArrayList(contents);
incorrectCopy.add(new ACartesianPoint(5,5));
Point firstElement = incorrectCopy.get(0);
List<Point> checkedCopy1 = new ArrayList<String>(contents);
List<Point> checkedCopy2 = new ArrayList<Point>(contents);
```

ClassCastException

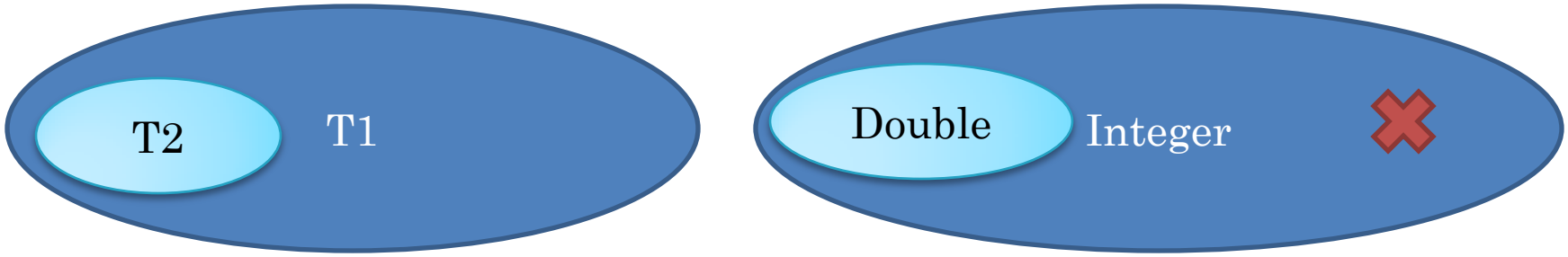
Return type, variable mismatch

Constructor formal/actual argument mismatch.

If creating calling constructor that takes no formal argument typed using type parameters, untyped instantiation is safe.

Extra Slides

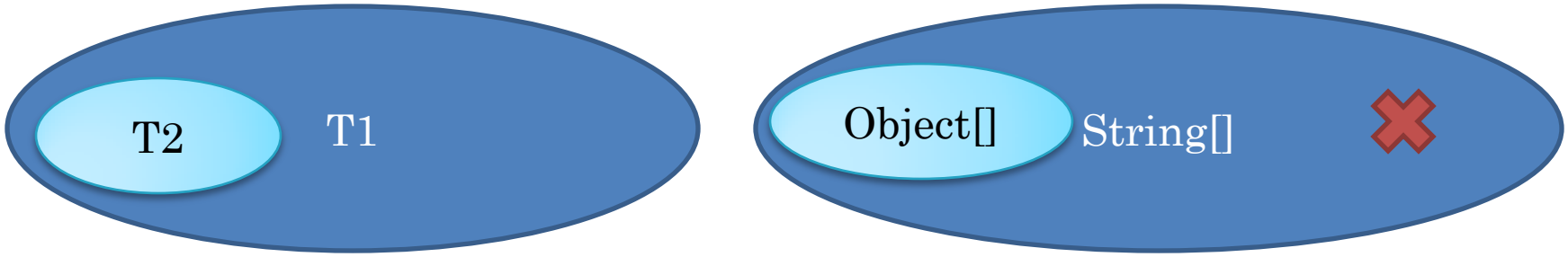
CounterIntitive IS-A



T1 IS-A T2 implies instances of T1 super set of instances of T1

Integer IS-A Double

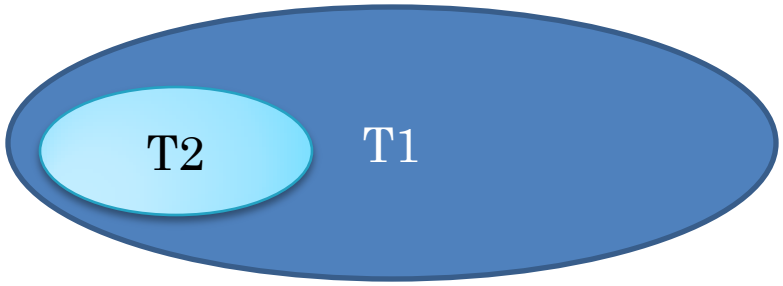
CounterIntitive IS-A

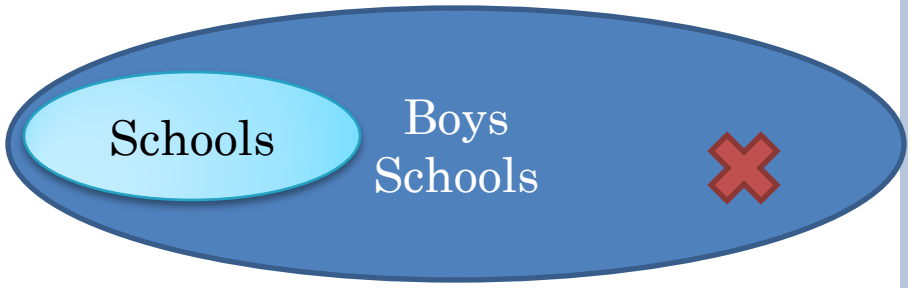


T1 IS-A T2 implies instances of T1 super set of instances of T1

String[] IS-A Object[]

CounterIntitive IS-A





T1 IS-A T2 implies instances of T1 super set of instances of T1

A girls(boys) school IS-A school

Cannot add a boy to a girls school.

Type Rules

```
String[] strings = new String[50];
Point[] points   = new String[50];
points = strings;
```

```
Object[] objects = new String[50];
```

```
objects[0] = new ACartesianPoint(4, 5);
```

Runtime ArrayStoreException because array assigned to objects not known at compile time

Should actual array have same type as variable?