Instructor: Prasun Dewan
PREREQUISITE

- Inheritance
- Collections
TOPICS

- **Generics**
  - Code that works for a multiple types.
  - Not related by IS-A relationships.
  - Not code working on Object, specific type information is not lost, requires no cast
  - Array IS-A relation

- **Adapter:**
  - Converter class sitting between two other classes.
  - Special case of delegator
**Duplication of Code in Collections**

- Collection interfaces and classes had so much similarity.
- How to get rid of code duplication?
public interface StringHistory {
    public void addElement(String element);
    public int size();
    public String elementAt(int index);
}

How to get rid of duplication in interface definitions?

public interface PointHistory {
    public void addElement(Point p);
    public Point elementAt(int index);
    public int size();
}
public interface ObjectHistory {
    public void addElement(Object element);
    public int size();
    public Object elementAt(int index);
}

Can add both objects and points to history!
History Collections

```java
public interface StringHistory {
    public void addElement(String element);
    public int size();
    public String elementAt(int index);
}

public interface PointHistory {
    public void addElement(Point p);
    public Point elementAt(int index);
    public int size();
}
```

How to get rid of duplication in interface definitions?
public double calculateMyCurrentBMI() {
    return 75/(1.77*1.77);
}

public double calculateMyFutureBMI() {
    return 77/(1.77*1.77);
}

public double calculateBMI(double weight, double height) {
    return weight/(height*height);
}

calculateBMI(75, 1.77);
calculateBMI(77, 1.77);

METHODS ANALOGY

How to get rid of duplication in method definitions?

Relace int constant with int parameter/placemarker

Only one implementation exists
**Duplication in Type Definitions**

```java
public interface StringHistory {
    public void addElement(String element);
    public int size();
    public String elementAt(int index);
}
```

```java
public interface PointHistory {
    public void addElement(Point p);
    public Point elementAt(int index);
    public int size();
}
```

Need to replace type constants with type parameter
public interface StringHistory {
    public void addElement(String element);
    public int size();
    public String elementAt(int index);
}

public interface PointHistory {
    public void addElement(Point p);
    public Point elementAt(int index);
    public int size();
}
**Java Generic Types with Type Parameters**

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

Can accidentally call say <t> or <aT> and Java will not catch error

Need a way to declare type parameter like we declare a variable
**Java Generic Types with Type Parameters**

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.

```java
public interface History<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.

- `History<String> stringHistory;`
- `History<Point> pointHistory;`

Assigning values to type parameters is a compile time activity for type checking only. The value assigned are “erased” at runtime.

How to indicate a string or point history?

How many versions of History at runtime?
### Java Generic Types with Type Parameters (Review)

#### Public Interface `History<T>`

- `public void addElement(T t);`
- `public T elementAt(int index);`
- `public size();`

#### Questions

- How to indicate a string or point history?
- How many versions of `History` at runtime?
- When is parameter assigned?

#### Examples

- `public interface History<T> {`...
- `History<String> stringHistory;`  
  `History<Point> pointHistory;`

#### Notes

- 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.
- A method, class, interface with type parameter is called a generic.
- Creating an elaboration of generic by giving actual value to type parameter.
- A single version in which `T == Object` is shared by all of its elaborations.
- Assigning values to type parameters is a compile time activity for type checking only. The value assigned are “erased” at runtime.
Arrays vs. Generics

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

History<String> stringHistory;
History<Point> pointHistory;
Double<Point> doubleHistory;

Generics and arrays are parameterized types

String[] strings;
Point[] points;
double[] doubles;

Are arrays generic with special syntax?

A single version in which T == Object is shared by all of its elaborations.
Assigning values to type parameters is a compile time activity for type checking only. The value assigned are “erased” at runtime

Different versions describe elements of different sizes.
Element type kept at runtime
public class AStringHistory implements History<String> {
    public final int MAX_SIZE = 50;
    String[] contents = new String[MAX_SIZE];
    int size = 0;
    public int size() {return size;}
    public String elementAt (int index) {
        return contents[index];
    }
    boolean isFull() {return size == MAX_SIZE;}
    public void addElement(String element) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            contents[size] = element;
            size++;
        }
    }
}
public class APointHistory implements History<Point> {
    public final int MAX_SIZE = 50;
    Point[] contents = new Point[MAX_SIZE];
    int size = 0;
    public int size() {return size;}
    public Point elementAt (int index) {
        return contents[index];
    }
    boolean isFull() {return size == MAX_SIZE;}
    public void addElement(Point element) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            contents[size] = element;
            size++;
        }
    }
}
public class AStringHistory implements History<String> {
    public final int MAX_SIZE = 50;
    String[] contents = new String[MAX_SIZE];
    int size = 0;
    public int size() {return size;}
    public String elementAt (int index) {
        return contents[index];
    }
    boolean isFull() {return size == MAX_SIZE;}
    public void addElement(String element) {
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            contents[size] = element;
            size++;
        }
    }
}

Replace all occurrences of String with T?
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++;
        }
    }
}
INSTANTIATING A HISTORY

```java
History<String> stringHistory = new AHistory<String>();
History<Point> pointHistory = new AHistory<Point>();
```
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) {
        System.out.println(element.charAt(0));
        System.out.println(element.getX());
        System.out.println(element.toString());
        if (isFull())
            System.out.println("Adding item to a full history");
        else {
            contents[size] = element;
            size++;
        }
    }
}

Only object operations can be assumed because the only guarantee is that T will be elaborated with a subtype of Object.
public class AHistory<T> implements History<T> {
    ...
}

The parameter of AHistory is used as the parameter of History.

public interface History<T> {
    ...
}

The parameter of AHistory is used as the parameter of History.

public double calculateMyBMI(double weight) {
    return calculateBMI(weight, 1.77);
}

The parameter of calculateMyBMI is used as the parameter of calculateBMI

public double calculateBMI(double weight, double height) {
    return weight/(height*height);
}
DECLARED VS. PASSED NAME

```java
public class AHistory<AType> implements History<T> {
    ...
}
```

```java
public interface History<T> {
    ...
}
```

Passed name must be the same as declared name

```java
public double calculateMyBMI(double weight) {
    return calculateBMI(w, 1.77);
}
```

```java
public double calculateBMI(double w, double height) {
    return w/(height*height);
}
```
public class AHistory<AType> implements History<AType> {
    ...
}

public interface History<T> {
    ...
}

Caller can rename the parameter

public double calculateMyBMI(double weight) {
    return calculateBMI(weight, 1.77);
}

public double calculateBMI(double w, double height) {
    return w/(height*height);
}
public interface I1 <T>{
    ...
}

public interface I2 <T>{
    ...
}

public class C<T1, T2> implements I1 <T1>, I2<T2>{
    ...
}
public interface History<T> {
    public void addElement(T element);
    public T elementAt (int index);
    public int size();
}

public interface PointHistory extends History<String> {
}

public interface PointHistory {
    public void addElement (int x, int y);
    public Point elementAt (int index);
    public int size();
}
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++;
        }
    }
}

public interface PointHistory {
    public void addElement (int x, int y);
    public Point elementAt (int index);
    public int size();
}
Adding to Generic?

```java
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++;
        }
    }
    public void addElement(int anX, int aY)
    {
        addElement(new A CartesianPoint(x, y));
    }
}
```

Method makes no sense for non point histories

Can only pass to addElement a T
public class APointHistory extends AHistory<Point>
implements PointHistory{
    public void addElement(int x, int y) {
        addElement(new ACartesianPoint(x,y));
    }
}

public interface History<T> {
    public void addElement(T element);
    public T elementAt (int index);
    public int size();
}

Inheriting APointHistory?

Has both addElement methods!

public interface PointHistory {
    public void addElement (int x, int y);
    public Point elementAt (int index);
    public int size();
}
**In delegation**, delegating class C1 has a reference to reused delegate class C2, rather than C1 IS-A C2.
Pattern?

- Adapter is a class that sits between a client and adaptee class much like an adapter sits between two objects that need to interact with each other.
- Methods called in adaptee through adapter.
- Degree of adaptation undefined.
  - Assumed no extra functionality offered but some may be removed.
- Methods offered to client
  - Adapted name
  - Adapted parameters.
**Generic Java Collection Types**

```java
import java.util.ArrayList;

public class AStringHistory implements History<String> {
    List<String> contents = new ArrayList<String>();
    public void addElement(String s) { contents.add(s); }
    public String elementAt(int index) { return contents.get(index); }
    public int size() { return contents.size(); }
}

/* Exposing only history methods Adapter pattern often used with ArrayList and Vector */

import java.util.ArrayList;
public class AStringHistory extends ArrayList<String> implements History<String> {
    /* remove() not visible. */
}
```
public interface Table {
    // associates key with value, returning last value associated with key
    public final Object put (Object key, Object value);
    // returns last value associated with key, or null if no association
    public final Object get (Object key);
}
public interface Table <KeyType, ValueType> {
    // associates key with value, returning last value associated with key
    public final ValueType put (KeyType key, ValueType value);
    // returns last value associated with key, or null if no association
    public final ValueType get (Object key);
}
public static void main (String[] args) {
    Map aMap = new HashMap();
aMap.put("Nadal", 11);
aMap.put("Federer", 17);
aMap.put("Sampras", 14);
    System.out.println(aMap.get("Nadal");
    System.out.println(aMap.get("nadal");
aMap.put("Nadal", 13);
    System.out.println(aMap.get("Nadal");
ObjectEditor.edit(aMap);
}
```java
public static void main (String[] args) {
    Map<String, Integer> aMap = new HashMap<String, Integer>();
    aMap.put("Nadal", 10);
    aMap.put("Federer", 17);
    aMap.put("Sampras", 14);
    System.out.println(aMap.get("Nadal"));
    System.out.println(aMap.get("nadal"));
    aMap.put("Nadal", 11);
    System.out.println(aMap.get("Nadal"));
    ObjectEditor.edit(aMap);
}
```
**Observer Interfaces**

```java
public interface CounterObserver {
    public void update(Counter aCounter);
}
```

```java
public interface BMISpreadsheetObserver {
    public void update(BMISpreadsheet aBMISpreadsheet);
}
```

```java
public interface GenerictObserver<Observable> {
    public void update(Observable anObservable);
}
```
ELABORATING GENERIC OBSERVER

```java
public class ACounterObserver implements GenericObserver<Counter> {
    public void update(Counter aCounter) {
        System.out.println(aCounter.getValue());
    }
}
```

```java
public class ABMISpreadsheetObserver implements GenericObserver<BMISpreadsheet> {
    public void update(BMISpreadsheet aBMISpreadsheet) {
        System.out.println(aBMISpreadsheet.getBMI());
    }
}
```
**Multiple Implementations**

```java
public class ABMISpreadsheetAndCounterObserver implements GenericObserver<BMISpreadsheet>, GenericObserver<Counter> {
    public void update(BMISpreadsheet aBMISpreadsheet) {
        System.out.println(aBMISpreadsheet.getBMI());
    }
    public void update(Counter aCounter) {
        System.out.println(aCounter.getValue());
    }
}
```

Cannot implement the same generic twice with different parameters

A single interface method exists at run time

An interface method is mapped to a class method at runtime (dynamic dispatch)

It can be mapped to a single class method at run time

```java
GenericObserver<Counter> counterObserver = new ABMISpreadsheetAndCounterObserver();
counterObserver.update(counter);
```
EXTRA
Java Vectors, Array Lists, Enumerations

- Just like collections we defined
- Except they can store and iterate arbitrary objects
JAVA COLLECTION CLASSES

AbstractCollection

AbstractSet

AbstractList

AbstractSequentialList

Collection

Set

List

HashSet

Vector

ArrayList

LinkedList
public interface PointHistory {
    public void addElement (Point p);
    public Point elementAt (int index);
    public int size();
}

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.

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

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

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

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

A single implementation is shared by all of its elaborations.

Assigning values to type parameters is a compile time activity for type checking only.
**Generic with Type Parameter**

```java
public interface History <T> {
    public void addElement(T element);
    public int size();
    public T elementAt(int index);
}
```

How to indicate a string or point history?

```java
History<String> stringHistory;
History<Point> pointHistory;
```
public interface History<T> {
    public void addElement(T element);
    public int size();
    public T elementAt(int index);
}

History<String> stringHistory;
History<Point> pointHistory;
public class APointHistory implements PointHistory {
    History<Point> contents = new AHistory();
    public void addElement(int x, int y) {
        contents.addElement(new ACartesianPoint(x, y));
    }
    public Point elementAt(int index) {
        return contents.elementAt(index);
    }
    public int size() {
        return contents.size();
    }
}

In delegation, class C1 has a reference to reused class C2, rather than C1 IS-A C2
Passing Type Parameters

```java
public class AHistory<AType> implements History<AType> {
    ...
}
```

```java
public double calculateMyBMI(double w) {
    return w/(1.77*1.77);
}
```

```java
public double calculateBMI(double weight, double height) {
    return weight/(height*height);
}
```