CS2030S

Programming Methodology II

Lecture 05: Generic

QnA

Recap

Recap

Exception
- Create

Exception

```
new Circle(new Point(1, 1), -1); // negative radius?
```

Creating Your Own

InvalidCircleException.java

```
class InvalidCircleException extends IllegalArgumentException {
  private Point center;
  public InvalidCircleException(Point c, String message) {
    super(message);
    this.center = c;
  }
  @Override public String toString() {
    return "The circle centred at " + this.center + " cannot be created: " + getMessage();
  }
}
```

Recap

Exception
- *Create*
- *Throw*

Exception

```
new Circle(new Point(1, 1), -1); // negative radius?
```

Throwing Exception

Circle_Exception.java

```
class Circle {
  private Point c;
  private double r;
  public Circle(Point c, double r) throws InvalidCircleException {
    if (r <= 0) { throw new InvalidCircleException(c, "Radius not positive"); }
    this.c = c;
    this.r = r;
  }
}
```

Generics

Generics

Recap: Pairs

Defining Integer Pairs

Int_Pair_05.java

```
class IntPair {
  private int first;
  private int second;

  public IntPair(int first, int second) {
      this.first = first;
      this.second = second;
  }

  public int getFirst() { return this.first; }
  public int getSecond() { return this.second; }
}
```

Generics

Recap: Pairs

Using Integer Pairs

```
IntPair findMinMax(int[] array) {
  int min = Integer.MAX_VALUE; // stores the min
  int max = Integer.MIN.VALUE; // stores the max
  for (int i : array) {
      if (i < min)  {
          min = i;
      }
      if (i > max) {
          max = i;
      }
  }
  return new IntPair(min, max);
}
```

Generics

Recap

Recap: Pairs

Defining Double Pairs?

Defining String Pairs?

Defining Pairs of Different Types?

```
class Pair {
  private Object first;  // Root of class hierarchy
  private Object second; // Root of class hierarchy
  public Pair(Object first, Object second) {
      this.first = first;
      this.second = second;
  }
  public Object getFirst() { return this.first; }
  public Object getSecond() { return this.second; }
}
```

Generics

Recap: Pairs

Using Object Pairs

```
Pair p = new Pair("hello", 4);

Integer i = (Integer) p.getFirst();
// run-time ClassCastException
```

Generics

Generic Pair

Defining Generic Pair

Pair_v1.java

```
class Pair<S,T> {
  private S first;
  private T second;

  public Pair(S first, T second) {
      this.first = first;
      this.second = second;
  }

  public S getFirst() { return this.first; }
  public T getSecond() { return this.second; }
}
```

Generics

Generic Pair

Using Generic Pair

```
Pair<String,Integer> p = new Pair<String, Integer>("hello", 4);

Integer i = (Integer) p.getFirst();
// compile-time error
```

Generics

Generic Pair

Extending Generic Class

```
class DictEntry<T> extends Pair<String,T> {
  // code omitted
}
```

Generics

Generic Pair

Implementing Generic Interface

Generics

Generic Methods

```
class Main {
  public static <T> boolean contains(    T [] array,    T   obj) {
    for (   T   curr : array) {
      if (curr.equals(obj)) {
        return true;
      }
    }
    return false;
  }
}
```

```
String[] strArray = new String[] { "hello", "world" };
Main.<String>contains(strArray, 123); // type mismatch error
```

Generics

Bounded Type Parameters

```
public static <T extends GetAreable> T findLargest(T[] array) {
  double maxArea = 0;
  T maxObj = null;
  for (T curr : array) {
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
      maxObj = curr;
    }
  }
  return maxObj;
}
```

Generics

Comparable

Interface Comparable

Generics

Comparable

Comparable Pairs

Pair_v3.java

```
class Pair<S extends Comparable<S>,T> implements Comparable<Pair<S,T>> {
  // code omitted
  @Override
  public int compareTo(Pair<S,T> s1) {
      return this.first.compareTo(s1.first)
  }
  @Override
  public String toString() {
    return this.first + " " + this.second;
  }
}
```

Generics

Recap
Generic Pair
Generic Methods
Bound
Comparable
- *Interface*
- *Pair*
- *Sorting*

Comparable

Sorting via java.util.Arrays.sort

Generics

Comparable

Sorting via java.util.Arrays.sort

```
Object[] array = new Object[] {
  new Pair<String,Integer>("Alice", 1),
  new Pair<String,Integer>("Carol", 2),
  new Pair<String,Integer>("Bob", 3),
  new Pair<String,Integer>("Dave", 3),
};

java.util.Arrays.sort(array);

for (Object o : array) {
  System.out.println(o);
}
```

Type Erasure

Type Erasure

Specialization

Code Specialization

> Generate a new class for every new type argument
>
> *(C#, C++, Rust)*

```
Pair<String,Integer> p1;
Pair<Double,Double>  p2;
```

```
class Pair_String_Integer { }
class Pair_Double_Double  { }
```

Type Erasure

Specialization
Sharing

Code Sharing

> Erase type arguments and type parameters *after* type checking
>
> *(Java)*

Type Erasure

Specialization
Sharing
- Caller

Code Sharing

> Erase type arguments and type parameters *after* type checking
>
> *(Java)*

Caller Code

```
Pair<String,Integer> p = new Pair<String,Integer>("hello", 4);
Integer i =              p.getSecond();
```

Type Erasure

Specialization
Sharing
- Caller

Code Sharing

> Erase type arguments and type parameters *after* type checking
>
> *(Java)*

Type Erasure *(After Type Checking)*

```
Pair                  p = new Pair                  ("hello", 4);
Integer i = (Integer) p.getSecond(); // type cast added by compiler
```

Type Erasure

Specialization
Sharing
- *Caller*
- *Callee*

Code Sharing

> Erase type arguments and type parameters *after* type checking
>
> *(Java)*

Callee Code

```
class Pair<S,T> {
  private    S   first;
  private    T   second;
  // code omitted
  public     S   getFirst()  { return this.first;  }
  public     T   getSecond() { return this.second; }
}
```

Type Erasure

Specialization
Sharing
- *Caller*
- *Callee*

Code Sharing

> Erase type arguments and type parameters *after* type checking
>
> *(Java)*

Type Erasure *(After Type Checking)*

```
class Pair      {
  private Object first;
  private Object second;
  // code omitted
  public Object getFirst()  { return this.first;  }
  public Object getSecond() { return this.second; }
}
```

Type Erasure

Specialization
Sharing
- *Caller*
- *Callee*
- *Bounded*

Code Sharing

> Erase type arguments and type parameters *after* type checking
>
> *(Java)*

Bounded Generic

```
class Pair<S extends Comparable<S>,T> {
  private     S      first;
  private     T      second;
  // code omitted
  public      S      getFirst()  { return this.first;  }
  public      T      getSecond() { return this.second; }
}
```

Type Erasure

Specialization
Sharing
- *Caller*
- *Callee*
- *Bounded*

Code Sharing

> Erase type arguments and type parameters *after* type checking
>
> *(Java)*

Type Erasure *(After Type Checking)*

```
class Pair                              {
  private Comparable first;
  private Object     second;
  // code omitted
  public Comparable getFirst()  { return this.first;  }
  public Object     getSecond() { return this.second; }
}
```

Type Erasure

Limitations

No Type Information

```
Pair<String,Integer>[] pairArray;
Object[] objArray;

pairArray = new Pair<String,Integer>[2];
objArray = pairArray;

objArray[0] = new Pair<Double,Boolean>(3.14, true);
String str =          pairArray[0].getFirst();
```

Type Erasure

Specialization
Sharing
Limitations
- No Type Info

Limitations

Type Erasure *(After Type Checking)*

```
Pair                [] pairArray; // Pair[]
Object[] objArray;                // Object[]

pairArray = new Pair                [2]; // Pair[]
objArray = pairArray;                    // Pair[] <: Object[] (Covariance)

objArray[0] = new Pair                (3.14, true); // Pair <: Object
String str = (String) pairArray[0].getFirst();
```

Type Erasure

Specialization
Sharing
Limitations
- *No Type Info*
- *Array ✗ Generic*

Limitations

Array ✗ Generic

```
new Pair<String,Integer>[2]; // ERROR
new Pair<S,T>[2];            // ERROR
new T[2];                    // ERROR
```

Array<T>

Array<T>

Skeleton

Skeleton

Array_v3.java

```
class Array<T> {
  private T[] array;

  public Array(int length) {
    this.array = (T[]) new Object[length]; // WARNING! (but not error)
  }

  public void set(int index, T t) {
    this.array[index] = t;
  }
  public T get(int index) {
    return this.array[index];
  }
  public T[] getArray() {
    return this.array;
  }
}
```

Array<T>

Skeleton
Unsafe

Unsafe Usage

```
Array<String> strArray = new Array<String>(2);
Object[] objArray = strArray.getArray();
objArray[0] = 5;
String s = strArray.get(0);
```

Array<T>

Safe

Array_v4.java

```java
class Array<T> {
  private T[] array;
  public Array(int length) {
    // The only way we can put an object into array is through the method set() and we
    // only put object of type T inside. So it is safe to cast `Object[]` to `T[]`.
    @SuppressWarnings("unchecked")
    T[] temp = (T[]) new Object[length];
    this.array = temp;
  }
  public void set(int index, T t) {
    this.array[index] = t;
  }
  public T get(int index) {
    return this.array[index];
  }
}
```

Array<T>

Raw Types

Do NOT Use Raw Types

```
Array strArray = new Array(2);
```

```
public static void setArray(Array a) {
  a.set(0, 1234);
}
```

Exception

```
obj instanceof Array
// evaluated at run-time
// so no compile-time type information anyway
```

```
jshell> /exit
|    Goodbye
```