# Generics in Java

# Generics Overview

- Enable code reuse in classes/interfaces.
  - As methods have parameters which enable code reuse, generics enable the creation of classes/interfaces which can reuse the same code with different inputs.
- Generics enable more compile type security.
- Help eliminate explicit casting from code

# Why Generics? Example without

```
1  package edu.nyu.cs9053.generics;
2
3  /**
4   * User: blangel
5   * Date: 10/13/14
6   * Time: 9:26 AM
7   */
8  public class Gift {
9
10     private final Object value;
11
12     private final Double cost;
13
14     public Gift(Object value, Double cost) {
15         this.value = value;
16         this.cost = cost;
17     }
18
19     public Object getValue() {
20         return value;
21     }
22
23     public Double getCost() {
24         return cost;
25     }
```

# Why Generics? Example without (cont)

```
1  public class GiftGiver {
2
3      public static void main(String[] args) {
4          Computer computer = new Computer();
5          Gift giftToJon = new Gift(computer, 1500d);
6
7          Bicycle bicycle = new Bicycle();
8          Gift giftToBob = new Gift(bicycle, 500d);
9
10         Object jonGift = giftToJon.getValue();
11         // What's jonGift??
12         Object bobGift = giftToBob.getValue();
13         // What's bobGift??
14
15     }
16
17 }
```

# Why Generics? Example without (cont)

```
1  public class GiftGiver {
2
3      public static void main(String[] args) {
4          Computer computer = new Computer();
5          Gift giftToJon = new Gift(computer, 1500d);
6
7          Bicycle bicycle = new Bicycle();
8          Gift giftToBob = new Gift(bicycle, 500d);
9
10         Object jonGift = giftToJon.getValue();
11         Computer jonComputer = (Computer) jonGift;
12
13         Object bobGift = giftToBob.getValue();
14         Bicycle bobBicycle = (Bicycle) bobGift;
15
16         // But what if i inverted the values?
17         Computer computerFail = (Computer) bobGift;
18         Bicycle bicycleFail = (Bicycle) jonGift;
19     }
20
```

# Why Generics? Example without (cont)

```
1  public class ComputerGift {
2
3      private final Computer value;
4
5      private final Double cost;
6
7      public ComputerGift(Computer value, Double cost) {
8          this.value = value;
9          this.cost = cost;
10     }
11
12     public Computer getValue() {
13         return value;
14     }
15
16     public Double getCost() {
17         return cost;
18     }
19
```

# Why Generics? Example with

```
1  package edu.nyu.cs9053.generics;
2
3  /**
4   * User: blangel
5   * Date: 10/13/14
6   * Time: 10:21 AM
7   */
8  public class Gift<T> {
9
10     private final T value;
11
12     private final Double cost;
13
14     public Gift(T value, Double cost) {
15         this.value = value;
16         this.cost = cost;
17     }
18
19     public T getValue() {
20         return value;
21     }
22
23     public Double getCost() {
24         return cost;
```

# Why Generics? Code Reuse

```
1  public class GiftGiver {
2
3      public static void main(String[] args) {
4          Computer computer = new Computer();
5          Gift<Computer> giftToJon = new Gift<Computer>(computer, 1500d);
6
7          Bicycle bicycle = new Bicycle();
8          Gift<Bicycle> giftToBob = new Gift<Bicycle>(bicycle, 500d);
9
10         Computer jonGift = giftToJon.getValue();
11
12         Bicycle bobGift = giftToBob.getValue();
13     }
14
15 }
```

# Why Generics? Type Safety

```
 8  public class GiftGiver {
 9
10      public static void main(String[] args) {
11          Computer computer = new Computer();
12          Gift<Computer> giftToJon = new Gift<Computer>(computer, 1500d);
13
14          Bicycle bicycle = new Bicycle();
15          Gift<Bicycle> giftToBob = new Gift<Bicycle>(bicycle, 500d);
16
17          Computer jonGift = giftToBob.getValue();
18
19          Bicycle bobGift = giftToJon.getValue();
20      }
21
22  }
23
```

# Generics More in Depth

- A class or interface can have zero to many generic types.
- Types are defined by one or more letters.
  - By convention a single uppercase letter is used
- Types must be defined after the class/interface name surrounded within < > characters
- Types with generic types do not subtype
- Types can be defined to extend from or be super classes of other types.
  - E.g.; `<T extends Computer>` or `<T super Computer>`
- Types are erased at compilation (not available at runtime)

# Generics More in Depth (cont)

```
1  package edu.nyu.cs9053.generics;
2
3  /**
4   * User: blangel
5   * Date: 10/13/14
6   * Time: 10:49 AM
7   */
8  public interface Pair<F, S> {
9
10     F getFirst();
11
12     S getSecond();
13
14 }
```

# Generics More in Depth (cont)

- A class/interface with a generic type does not share the type hierarchy of its generic types.

```
1   public class Echo<T> {
2
3       public T echo(T value) {
4           return value;
5       }
6
7       public Echo<T> echo(Echo<T> value) {
8           return value;
9       }
10
11  }
```

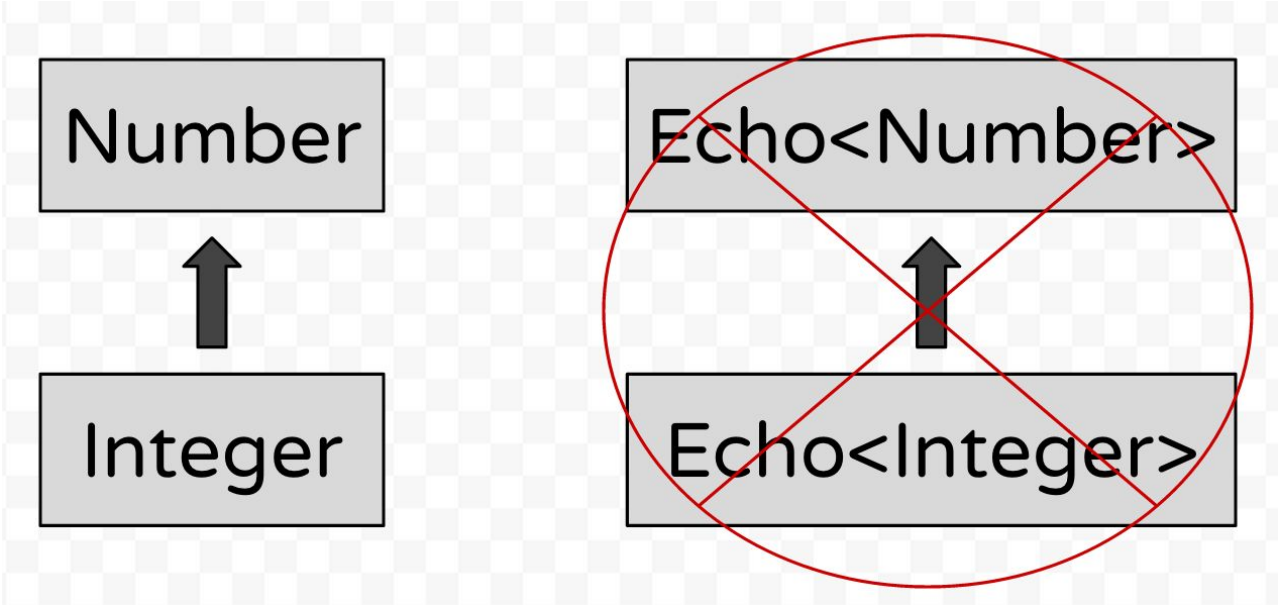
# Generics More in Depth (cont)

- Does this compile?

```
1  public class EchoChamber {
2
3      public static void main(String[] args) {
4          Echo<Number> numberEcho = new Echo<Number>();
5          numberEcho.echo(10); // echo(Integer)
6          numberEcho.echo(10d); // echo(Double)
7          numberEcho.echo(10f); // echo(Float)
8          numberEcho.echo(10L); // echo(Long)
9
10         numberEcho.echo(new Echo<Integer>());
11         numberEcho.echo(new Echo<Double>());
12         numberEcho.echo(new Echo<Float>());
13         numberEcho.echo(new Echo<Long>());
14     }
15
```

# Generics More in Depth (cont)

- It does not!



# Generics More in Depth (cont)

- Generic types can be bounded.  Subclass bound example:

```
1  public class BoundedEcho<T extends Number> {
2
3      public T echo(T value) {
4          return value;
5      }
6
7      public BoundedEcho<T> echo(BoundedEcho<T> value) {
8          return value;
9      }
10 }
```

# Generics More in Depth (cont)

- Does this compile?

```
1  public class BoundedEchoChamber {
2
3      public static void main(String[] args) {
4          BoundedEcho<Number> numberEcho = new BoundedEcho<Number>();
5          numberEcho.echo(10); // echo(Integer)
6          numberEcho.echo(10d); // echo(Double)
7          numberEcho.echo(10f); // echo(Float)
8          numberEcho.echo(10L); // echo(Long)
9
10         BoundedEcho<String> stringEcho = new BoundedEcho<String>();
11
12         numberEcho.echo(new BoundedEcho<Integer>());
13         numberEcho.echo(new BoundedEcho<Double>());
14         numberEcho.echo(new BoundedEcho<Float>());
15         numberEcho.echo(new BoundedEcho<Long>());
16     }
17
```

# Generics More in Depth (cont)

- A generic type can be bounded by multiple types. Only one of which can be a Class however.

```
1  public class MultipleBounds<T extends Number & Comparable & Serializable> {
2
3      private final T number;
4
5      public MultipleBounds(T number) {
6          this.number = number;
7      }
8
9      public T getNumber() {
10         return number;
11     }
12 }
```

# Generics More in Depth (cont)

- Does this compile?

```
1  public class MultipleBounds<T extends Comparable & Integer> {
2
3      private final T number;
4
5      public MultipleBounds(T number) {
6          this.number = number;
7      }
8
9      public T getNumber() {
10         return number;
11     }
12 }
```

# Generics More in Depth (cont)

- Generic types can be bounded by other generic types.

```
1  public class BoundedGenericTypes<T, S extends T> {
2
3      private final T value;
4
5      private final S subValue;
6
7      public BoundedGenericTypes(T value, S subValue) {
8          this.value = value;
9          this.subValue = subValue;
10     }
11
12     public T getValue() {
13         return value;
14     }
15
16     public S getSubValue() {
17         return subValue;
18     }
19 }
```

# Generics More in Depth (cont)

- The generic parameter defined on the Class/Interface isn't available in a static context.
- I.e., this **does not** compile

```
1  public class GenericsAreNotStatic<T> {
2
3      private static T reference;
4
5  }
```

# Generics More in Depth (cont)

- Generic types are not reified; i.e., after compilation they are removed and not available at runtime. This is also called type erasure.

```
1  public class RuntimeGenerics<T> {
2
3      public static void main(String[] args) {
4          RuntimeGenerics<Number> runtimeGenericNumber = new RuntimeGenerics<Number>(10);
5
6          // compiler inserts the following
7          // Number numberValue = (Number) runtimeGenericNumber.getValue();
8          Number numberValue = runtimeGenericNumber.getValue();
9
10         RuntimeGenerics<String> runtimeGenericString = new RuntimeGenerics<String>("foobar");
11         // compiler inserts the following
12         // String stringValue = (String) runtimeGenericString.getValue();
13         String stringValue = runtimeGenericString.getValue();
14
15     }
16
17     private final T value;
18
19     public RuntimeGenerics(T value) {
20         this.value = value;
21     }
22
23     public T getValue() {
24         return value;
25     }
```

# Generics More in Depth (cont)

- Because Java does not have reified generics:
  - Cannot use primitive types as generic types
    - No! `Gift<int>`
  - Cannot use ‘instanceof’ check for generically parameterized types
    - No! `gift instanceof Gift<Computer>`
  - Cannot make exception classes with generic types
    - No! `public class MyException<T> extends Exception`
  - Cannot have array types of generically parameterized types
    - No! `Gift<Computer>[]`

# Generics Defined at Methods

- Generic parameters can also be defined at a method level.
  - But not at a field level

```
1  public class GenericMethods<T> {
2
3      public <S extends T> T transform(S value) {
4          return value;
5      }
6
7  }
```

# Generics Defined at Methods (cont)

- Can be defined for static methods as well.
  - Note, static methods still do not have access to class generics.

```
1   public class GenericStaticMethods<T> {
2
3       public static <S extends Number> S echo(S value) {
4           return value;
5       }
6
7       // This does not compile
8   //      public static <S extends T> S echo(T value) {
9   //          return value;
10  //      }
11
12  }
```

# Generics and Inheritance

- When extending/implementing classes/interfaces with generic parameters you must respect the super-types restrictions.

```java
public class GenericClass<T> {

    private final T value;

    public GenericClass(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

```java
public class SubGenericClass<T extends Number> extends GenericClass<T> {

    public SubGenericClass(T value) {
        super(value);
    }

    @Override public T getValue() {
        return super.getValue();
    }
}
```

# Wildcards!

- Remember that `Echo<Integer>` is not an instance of type `Echo<Number>` (whereas `Integer` is an instance of type `Number`)? Keep that in mind and look at the following code?
- Non-generic Gift printer ->

```
1  public class GiftPrinter {
2
3      public void print(Gift gift) {
4          System.out.printf("%s%n", gift);
5      }
6
7  }
```

# Wildcards! (cont)

- Now look at the generics Gift printer

```
1  public class GiftPrinter {
2
3      public void print(Gift<Object> gift) {
4          System.out.printf("%s%n", gift);
5      }
6
7  }
```

# Wildcards! (cont)

- Oh uh...what's wrong?

```
11  public class GiftPrinter {
12
13      public static void main(String[] args) {
14          Gift<Computer> computerGift = new Gift<Computer>(new Computer(), 1500d);
15          GiftPrinter printer = new GiftPrinter();
16          printer.print(computerGift);
17      }
18
19      public void print(Gift<Object> gift) {
20          System.out.printf("%s%n", gift);
21      }
22
23  }
24
```

# Wildcards! (cont)

- We've made `GiftPrinter` take a `Gift` of generic type `Object` and as we know, `Gift<Computer>` does not extend from `Gift<Object>`
- What we want is the generically typed Gift which is the supertype of all other generically typed Gift types. In Java, this is called the wildcard type!

```
11  public class GiftPrinter {
12
13      public static void main(String[] args) {
14          Gift<Computer> computerGift = new Gift<Computer>(new Computer(), 1500d);
15          GiftPrinter printer = new GiftPrinter();
16          printer.print(computerGift);
17      }
18
19      public void print(Gift<?> gift) {
20          System.out.printf("%s%n", gift);
21      }
22
23  }
```

# Wildcards! (cont)

- Wildcard types can only be used on instances (not class or methods)
  - You **cannot** do `public class Type<?> {`
  - You **cannot** do `public <?> void methodName() {`
  - You **cannot** do `public ? methodName() {`
  - You **can** do `public void methodName(Gift<?> gift) {`
- Only objects can use wildcard type parameters (variables, method parameters, etc).

# Wildcards! (cont)

- Bounded wildcards (? extends Type)

```
1  public class BoundedWildcard {
2
3      public void foo(Gift<? extends Number> gift) {
4          //
5      }
6
7  }
```

# Wildcards! (cont)

- The super bound! <? super X>
  - Whereas <? extends X> means a type which extends (is a subtype of) X the <? super X> means a type which is a super class of X

```
1  public class BoundedWildcard {
2
3      public void subClasses(Gift<? extends Number> gift) {
4
5      }
6
7      public void superClasses(Gift<? super Integer> gift) {
8
9      }
10
11 }
```

# Wildcards! (cont)

Can be confusing and a bit intimidating when using. The best written explanation of wildcards in the Java language I've found is Angelika Langer's discussion of Java Generics.

- http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html

A nice mnemonic for keeping straight extends and super is PECS -> **producer extends consumer super**

- If you need something to read of type T, then use <? extends T>
  - Collection<? extends T> can read values as type T
- If you need something to consume of type T, then use <? super T>
  - Collection<? super T> can write values into the collection as type T

# Read Chapter 9

All sections and also read 5.3 (ArrayList)

# Homework 7

https://github.com/NYU-CS9053/Fall-2016/homework/week7