JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

1. Classes and Instances
2. The Methods
3. The Constructors
4. Static Elements
5. Initialization sections
6. Package
7. Inheritance and Polymorphism
8. Abstract classes and Interfaces
9. String processing
10. Wrapper classes for primitive types
11. Exceptions and Assertions
12. Nested classes
13. Enums
14. **Generics**
15. Collections
16. Method overload resolution
17. Multithreads
18. Core Java classes
19. Object Oriented Design
20. Functional Programming

Module contents

- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problem
  - Wildcards. extends and super keywords

Module contents

- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Generics and JVM
  - Resriction of generic types
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problemm
  - Wildcards. extends and super keywords

The Generics 1/4

List<E> list

- Java 5 introduces *generics*, which allows us to *abstract over types* (or *parameterized types*).
- The class designers can be *generic about types in the definition*, while the users can be *specific in the types during the object instantiation or method invocation*.
- *Generics* make it possible to create classes (interfaces), methods (constructors) in such a way that *they can operate with different data types without implementing* ***several different*** classes (interfaces), methods (constructors).

The Generics 2/4

Generics are described as follows:

- Provide compile-time type safety
- Eliminate the need for casts
- Provide the ability to create compiler-checked homogeneous collections

Examples:

List<E> list

static <T> void sort(List<T> list, Comparator<? super T> c)

The Generics 3/4

- A generic class is defined with the following format:

  class name<T1, T2, ..., Tn> { /* ... */ }

  public class MyClass<T>

- The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

```
public class Main {
    public static void main(String[] args) {
        MyClass<String> strings = new MyClass<>();
    }
}
```

T1, T2, ... Tn – type parameters (variables)

<String>, <Integer>, ... - type arguments

The Generics 4/4

Formal Type Parameter Naming Convention

- Use an uppercase single-character for formal type parameter. For example,
- <E> for an element of a collection;
- <T> for type;
- <K, V> for key and value.
- <N> for number
- S,U,V, etc. for 2nd, 3rd, 4th type parameters

Module contents

- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problemm
  - Wildcards. extends and super keywords

Declaring and using generic types 1/7

```
1. public class MyBox {
2.     private Object object;
3.     public void set(Object object) {
4.         this.object = object;
5.     }
6.     public Object get() {
7.         return object;
8.     }
9. }
```

Declaring and using generic types 2/7

```
1. MyBox mb1 = new MyBox();
2. mb1.set(Integer.valueOf(10));
3. Integer x1 = (Integer)mb1.get(); // OK!     type casting
4. mb1.set("Hello!");                          // is necessary
5. Integer x2 = (Integer)mb1.get(); // Exception!
```

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
cannot be cast to java.lang.Integer
        at
com.brainacad.oop1.testgenerics.Main.main
```

Declaring and using generic types 3/7

- Generic type declaration

```
1.  public class MyBox<T> {
2.      // T stands for "Type"
3.      private T t;
4.      public void set(T t) {
5.          this.t = t;
6.      }
7.      public T get() {
8.          return t;
9.      }
10. }
```

T - type parameter
(or type variable)

Declaring and using generic types 4/7

- A generic type invocation, which replaces T with some concrete value, such as Integer

Integer - type argument

```
1. MyBox<Integer> mb1 = new MyBox<Integer>();
2. mb1.set(Integer.valueOf(10));
3. Integer x1 = mb1.get(); // OK!
4. mb1.set("Hello!");
5. Integer x2 = mb1.get();
```

no casting needed

Compile error!

Declaring and using generic types 5/7

- In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context

- MyBox<Integer> mb1 = **new** MyBox<>();

<> - diamond operator

Declaring and using generic types 6/7

- A generic class or interface can have multiple type parameters

generic interface

```
1. interface Pair<K, V> {
2.     public K getKey();
3.     public V getValue();
4. }
```

Declaring and using generic types 7/7

generic class implementing generic interface

```
1.  class OrderedPair<K, V> implements Pair<K, V> {
2.      private K key;
3.      private V value;
4.      public OrderedPair(K key, V value) {
5.          this.key = key;
6.          this.value = value;
7.      }
8.      public K getKey()  { return key; }
9.      public V getValue() { return value; }
10. }
```

Using generic class implemented generic interface

```
public static void main(String[] args) {
   OrderedPair<Integer, Character> charPair =
                                        new OrderedPair<>(0, 'A');
   OrderedPair<Integer, Double> doublePair =
                                        new OrderedPair<>(1, Math.PI);
   System.out.printf("charPair's key= %d and value  =
           %c%n", charPair.getKey(), charPair.getValue());
   System.out.printf("doublePair's key= %d and value =
          %f%n", doublePair.getKey(), doublePair.getValue());
}
```

Module contents

- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Bounded Type Parameters
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problemm
  - Wildcards. extends and super keywords

Declaring and using generic methods 1/4

```
1.  public class Main {
2.      public static void main(String[] arg) {
3.          Integer[] arr1 = {1, 2, 3, 4, 5};
4.          Integer res = (Integer)testGenerics(arr1, 2);
5.          //...
6.      }
7.      public static Object testGenerics(Object[] a, int x) {
8.          Object someResult = a[x];
9.          //...
10.         return someResult;
11.     }
12. }
```

type cast required

non-generic method

Declaring and using generic methods 2/4

```
1.  public class Main {
2.      public static void main(String[] arg) {
3.          Integer[] arr1 = {1, 2, 3, 4, 5};
4.          Integer res = testGenerics(arr1, 2);
5.          //...
6.      }
7.      public static <T> T testGenerics(T[] a, int x) {
8.          T someResult = a[x];
9.          //...
10.         return someResult;
11.     }
12. }
```

no type cast required

generic method with the type parameter T

introduces type parameter in the declaration of the method

generic method - generic algorithm!

Generic class with generic method

```
public class MyBox<T> {
    private final T t;
    public MyBox(T t) {
        this.t = t;
    }
    public <T> T tricky(T t) {
        return t;
    }
    @Override
    public String toString() {
        return "MyBox{" + "t=" + t + '}';
    }
}
```

A class and a method can have the same variable of type, but corresponding arguments of types can be different.

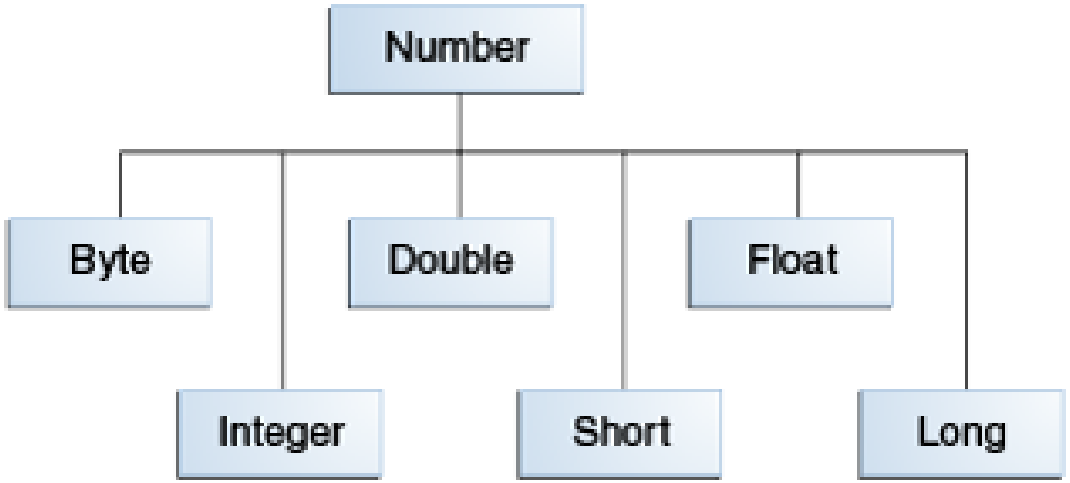
```
MyBox<String> mb = new MyBox<>("Hello");       //T is String
System.out.println(mb.tricky(LocalDate.now())); //T is LocalDate
```

Module contents

- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Bounded Type Parameters
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problemm
  - Wildcards. extends and super keywords

Bounded Type Parameters 1/6

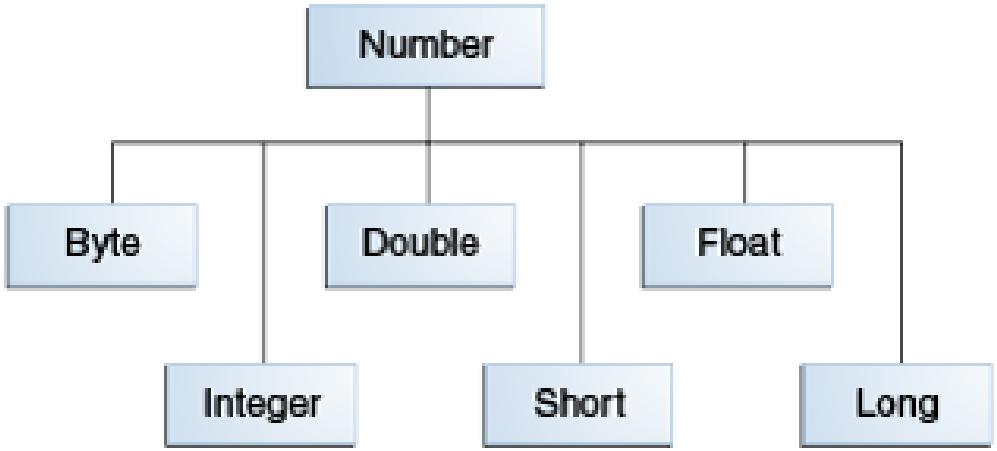
```
1.  Upper bound type parameter
2.  public class MyBox<T extends Number> {
3.      // T must by Number or its subtype
4.      private T t;
5.      public void set(
6.          this.t = t;
7.      }
8.      public T get() {
9.          return t;
10.     }
11. }
```



Bounded Type Parameters 2/6

1. MyBox<Integer> mb1 = **new** MyBox<>();
2. mb1.set(Integer.*valueOf*(10));
3. MyBox<Double> mb2 = **new** MyBox<>();
4. mb2.set(Double.*valueOf*(5.5));
5. MyBox<String> mb3 = **new** MyBox<>();

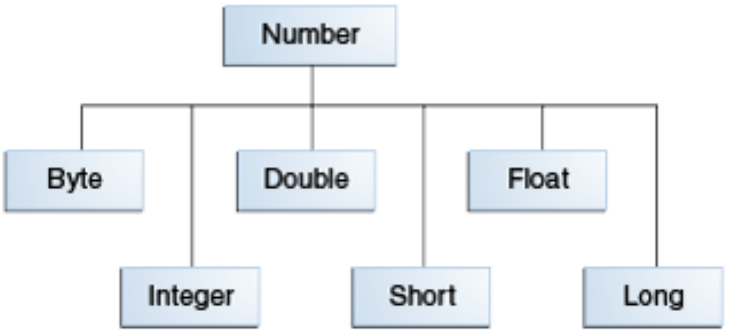
T must by Number or its subtype



Bounded Type Parameters 3/6

- Bounded type parameters allow you to invoke methods defined in the bounds type

```
1. public class MyBox<T extends Number> {
2.     // T must by Number or its subtype
3.     private T t;
4.     // ...
5.     public boolean isEven() {
6.         return t.intValue() % 2 == 0;
7.     }
8. }
```



Bounded Type Parameters 4/6

```
1. MyBox<Integer> mb1 = new MyBox<>();
2. mb1.set(Integer.valueOf(10));
3. System.out.println(mb1.isEven());     //true
4. MyBox<Double> mb2 = new MyBox<>();
5. mb2.set(Double.valueOf(5.5));         //false
6. System.out.println(mb2.isEven());
```

Bounded type generic method

Bounded Type Parameter

```
class Util {
   public <T extends Comparable<T>> int countGreaterThan(T[] arr,
                                                          T maxelm) {
      int count = 0;
      for (T elm : arr) {
         if (elm.compareTo(maxelm) > 0) {
            count++;
         }
      }
      return count;
   }
}
```

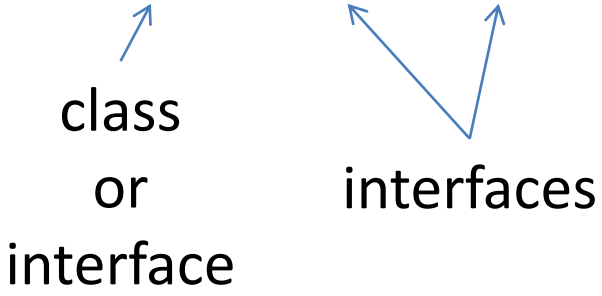
Bounded type generic method using

```
class Main {
    public static void main(String[] args) {
        Util util = new Util();
        String[] strArr = {"abc", "if", "no", "yes"};
        Integer[] intArr = {1, 3, 2, 5, 4};

        System.out.println("Number of elements > \"if\" = "
                                + util.countGreaterThan(strArr, "if"));
        System.out.println(" Number of elements > 2 = "
                                + util.countGreaterThan(intArr, 2));
    }
}
```

Bounded Type Parameters 5/6

- Multiple Bounds
- The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:
- <T extends B1 & B2 & B3>



Bounded Type Parameters 6/6

- A type variable with multiple bounds

1. **class** A { /* … */}
2. **interface** B { /* … */}
3. **interface** C { /* … */}
4. **class** D <T **extends** A & B & C> { /* … */}

Module contents

- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problemm
  - Wildcards. extends and super keywords

Generics and JVM 1/2

- Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming.
- To implement generics, the Java compiler applies **type erasure** - the Java compiler ***does not create different versions of a generic class for different type arguments***. ***It deletes all information about generic types (change it to Object)*** and performs the necessary typecasting operations to make the behavior of the application code look like a specific version of the generic class has been created.

Generics and JVM 2-2

- Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types

Type erasure revelation

```
MyBox<Integer> integerBox = new MyBox<>();
MyBox box = integerBox;          //Row type
MyBox<String> stringBox = box;
stringBox.set("Hello world");
System.out.println("Integer value = " + integerBox.get());
```

Output:

Integer value = Hello world

```
public class MyBox<T> {
    private T t;
    public T get() {
        return t;
    }
    public void set(T t) {
        this.t = t;
    }
}
```

Declaring and using generic types - **Row Type**

- A ***row type*** is the name of **generic** class or interface without any type arguments

```
45: ldc2_w        #40                 // double 5.5d
48: invokestatic  #42                 // Method java/lang/Double.valueOf:(D)Ljava/lang/Double;
51: invokevirtual #16                 // Method generics/typeerasure/MyBox.set:(Ljava/lang/Object;)V
54: getstatic     #20                 // Field java/lang/System.out:Ljava/io/PrintStream;
57: aload_2
58: invokevirtual #26                 // Method generics/typeerasure/MyBox.get:()Ljava/lang/Object;
61: invokevirtual #47                 // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
 11: invokestatic  #10                // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
 14: invokevirtual #16                // Method generics/typeerasure/MyBox.set:(Ljava/lang/Object;)V
 17: getstatic     #20                // Field java/lang/System.out:Ljava/io/PrintStream;
 20: aload_1
 21: invokevirtual #26                // Method generics/typeerasure/MyBox.get:()Ljava/lang/Object;
 24: checkcast     #11                // class java/lang/Integer
 27: invokevirtual #30                // Method java/lang/Integer.intValue:()I
 30: bipush        10
 32: iadd
 33: invokevirtual #34                // Method java/io/PrintStream.println:(I)V
```

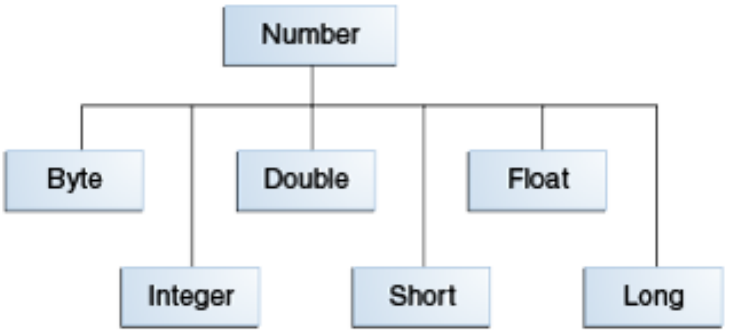
```
//    System.out.println(mb2.get() + 5.5);
    }
```

Repeat

Bounded Type Parameters 3/6

- Bounded type parameters allow you to invoke methods defined in the bounds type

```
1. public class MyBox<T extends Number> {
2.     // T must by Number or its subtype
3.     private T t;
4.     // ...
5.     public boolean isEven() {
6.         return t.intValue() % 2 == 0;
7.     }
8. }
```



Repeat

Bounded Type Parameters 4/6

1. MyBox<Integer> mb1 = **new** MyBox<>();
2. mb1.set(Integer.*valueOf*(10));
3. System.***out***.println(mb1.isEven());
4. MyBox<Double> mb2 = **new** MyBox<>();
5. mb2.set(Double.*valueOf*(5.5));
6. System.***out***.println(mb2.isEven());

```
 9: bipush        10
11: invokestatic  #10                 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
14: invokevirtual #16                 // Method generics/boundedtype/MyBoundedBox.set:(Ljava/lang/Number;)V
17: getstatic     #20                 // Field java/lang/System.out:Ljava/io/PrintStream;
20: aload_1
21: invokevirtual #26                 // Method generics/boundedtype/MyBoundedBox.get:()Ljava/lang/Number;
24: invokedynamic #30,  0             // InvokeDynamic #0:makeConcatWithConstants:(Ljava/lang/Number;)Ljava/lang/String;
29: invokevirtual #34                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

Module contents

- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problemm
  - Wildcards. extends and super keywords

Restrictions on Generics 1/8

1. Type arguments cannot be primitive types
2. You cannot create instances of type parameters
3. You cannot declare static fields with types that is type parameters
4. You cannot cast to or use instanceof with type parameters
5. You cannot create arrays of types that is type parameters (but You can declare such arrays in generic classes and generic methods)
6. You cannot catch or throw exceptions of type parameters (but You can declare such exception in method throws clause if methods belongs to the generic class with type parameter T extends Throwable or it descendants)
7. You cannot overload generic methods with the same methods with different type arguments
8. You cannot implement the same generic interfaces with different type arguments

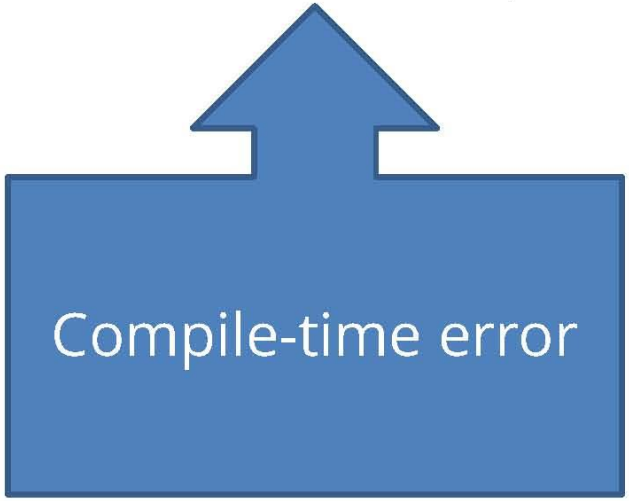
most restrictions - due to type erasure

Restrictions on Generics 2/8

- Type arguments cannot be primitive types

1. `Pair<Integer, Character> p2 = new OrderedPair<>(12, 'k'); // OK!`

2. `Pair<int, char> p1 = new OrderedPair<>(12, 'k');`

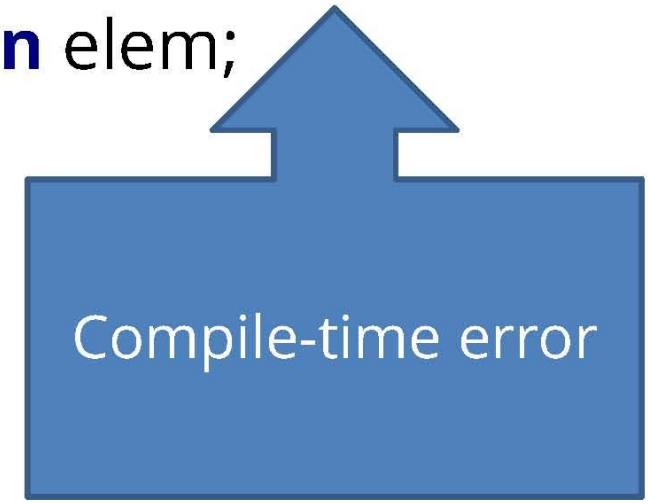


unexpected type
  required: reference
  found: int

Restrictions on Generics 3/8

- You cannot create instances of type parameters

```
public static <E> E test() {
    E elem = new E();
    return elem;
}
```



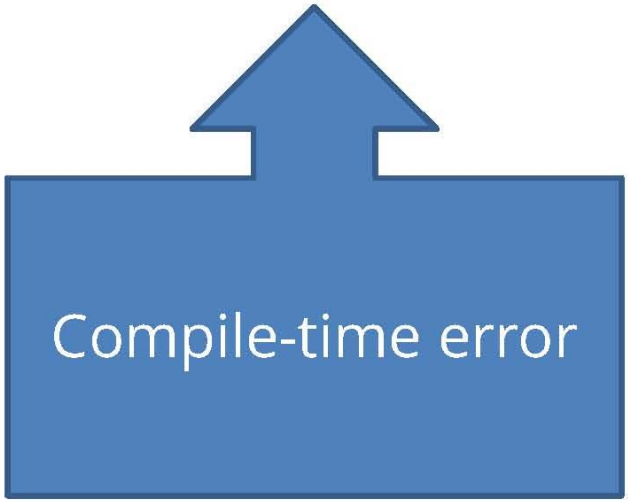
unexpected type
required: class
found: type parameter E
where E is a type-variable:
E extends Object declared
in method <E>test()

Restrictions on Generics 4/8

- You cannot declare static fields with types that is type parameters

static fields can be used
by different objects
of a generic class
with different type arguments

```
1. class MyClass<T> {
2.     private static T mos;
3.     // ...
4. }
```



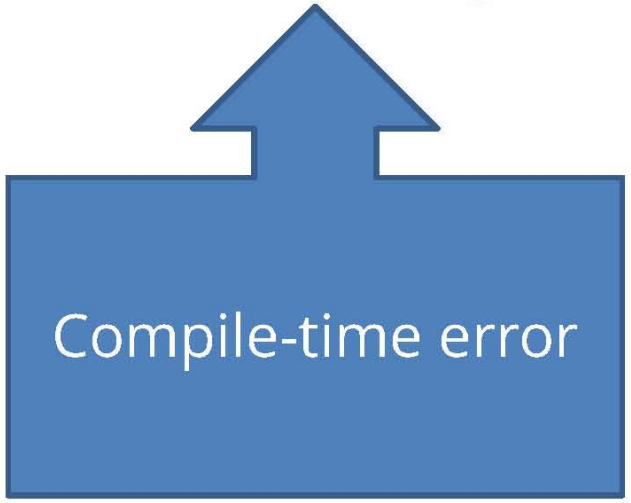
non-static type variable T
cannot be referenced
from a static context

Restrictions on Generics 5/8

- You cannot cast to or use instanceof with type parameters

illegal generic type for instanceof

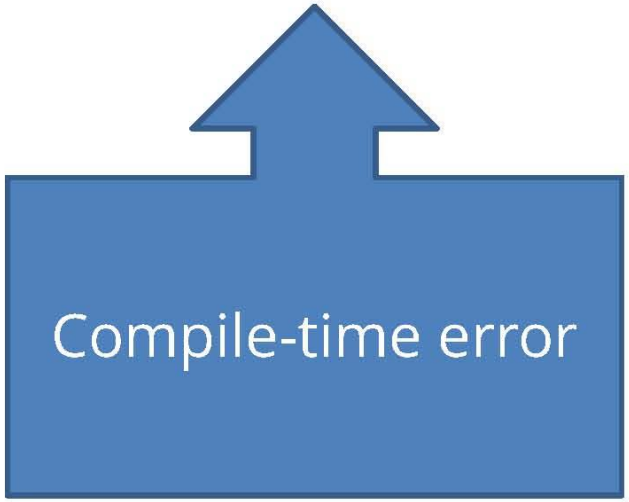
```
1. public static <K, V> void test(Pair<K, V> pr) {
2.     if (pr instanceof OrderedPair<Integer, String>) {
3.         // ...
4.     }
5. }
```



Restrictions on Generics 6/8

- You cannot create arrays of types that is type parameters (but You can declare such arrays in generic classes and generic methods)

```
1. Pair<Integer, Character>[] parr1 = new
2.         OrderedPair<Integer, Character>[10];
```



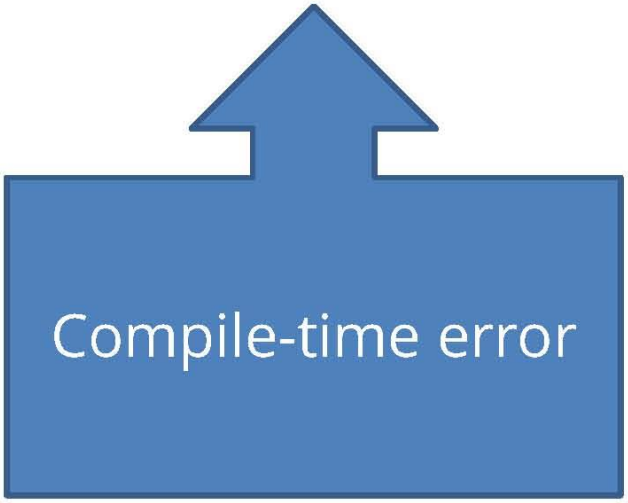
generic array creation

Restrictions on Generics 7/8

- You cannot catch or throw exceptions of type parameters (but You can declare such exception in method throws clause if methods belongs to the generic class with type parameter T extends Throwable or it descendants)

```
1. class MathException<T> extends Exception {
2.  //...
3. }
```

a public class cannot extend java.lang.Throwable



Cannot catch type parameter example

```
public class GenericsException {
   public static <T extends Exception, J> void execute(List<J>
jobs) {
      try {
         for (J job : jobs) {
            System.out.println(job);
         }
       } catch (T e) {
         ...
      } finally {
         jobs.clear();
         throw new T();
      }
   } }
```

Compile-time error:
unexpected type
 required: class
 found: type parameter T
 where T,J are type-variables:
  T extends Exception declared in method
  <T,J>execute(List<J>)
  J extends Object declared in method
  <T,J>execute(List<J>)

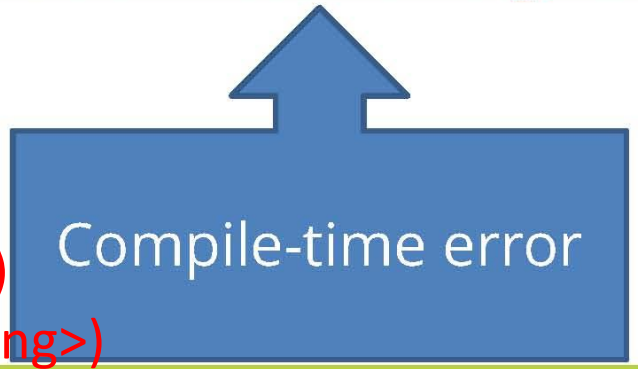
Methods can throws type parameter example

```
class Parser <T extends Throwable> {
    public void parse(File file) throws T {      // OK
        // ...
    }
}
```

Restrictions on Generics 8/8

- You cannot implement the same generic interfaces with different type arguments

```
1. class MyExample {
2.     public void test(OrderedPair<Integer, String> a) {
3.         //...
4.     }
5.     public void test(OrderedPair<String, Double> a) {
6.         //...
7.     }
8. }
```



name clash:
test(OrderedPair<Double,String>)
and test(OrderedPair<Integer,String>)
have the same erasure

Cannot implement two same interfaces with different type parameters

```
public class DecimalString implements Comparable<Number>,
                                      Comparable<String> {}
```

Compile-time error:

repeated interface

- Comparable cannot be inherited with different arguments:
  <java.lang.String> and <java.lang.Integer>

Module contents

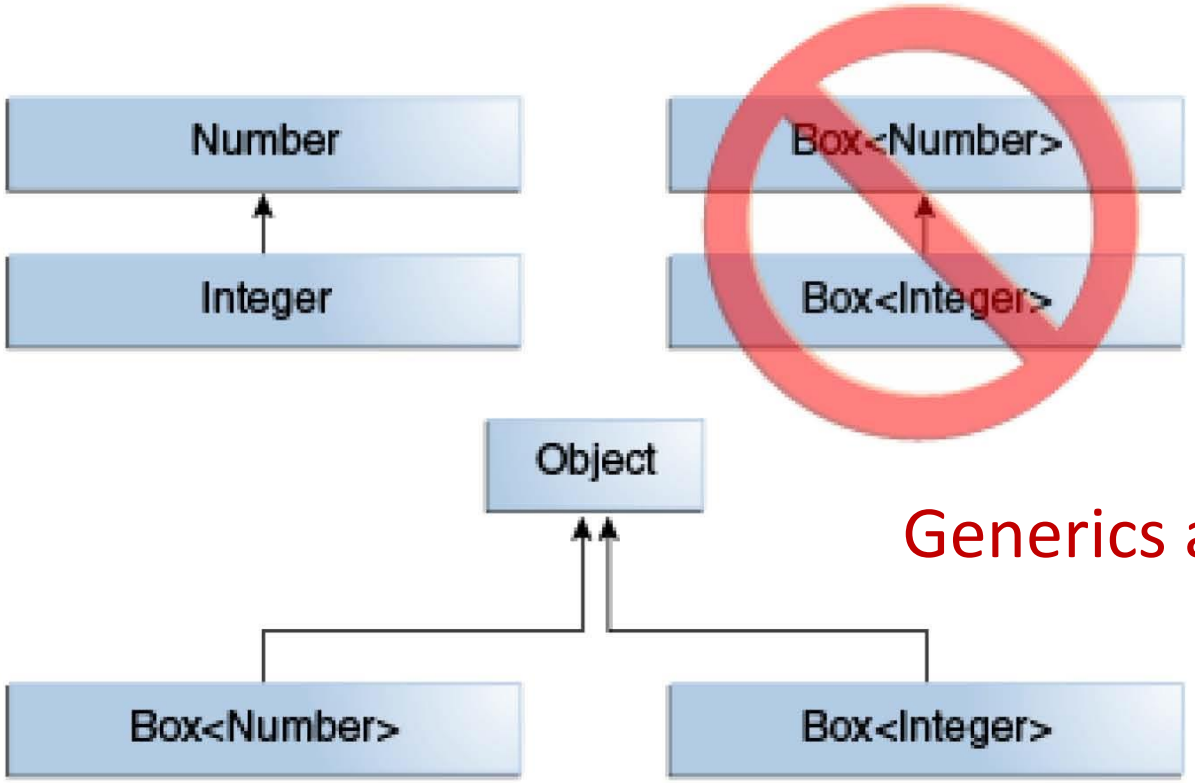
- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problemm
  - Wildcards. extends and super keywords

Generic and inheritance 1/2

- Given two concrete types A and B (for example, Number and Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related.
- The common parent of MyClass<A> and MyClass<B> is Object.

Generic and inheritance 2/2

- Box<Integer> is not a subtype of Box<Number> even though Integer is a subtype of Number



Generics inheritance investigation

```
public class GenericInherit {
    static class MyClass<T> {
    }
    public static void main(String[] args) {
        String str = "abc";     //String extends Object
        Object obj = new Object();
        obj = str;   // works because String is-a Object, inheritance in Java
        MyClass<String> myClass1 = new MyClass<String>();
        MyClass<Object> myClass2 = new MyClass<Object>();
        myClass2 = myClass1; // compilation error since MyClass<String>
                             // is not a MyClass<Object>
       myClass1 = (MyClass<String>)myClass2; //the same error - generics
                                             // can not be casted
        obj = myClass1;        // MyClass<T> parent is Object
    }
}
```

Module contents

- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problemm
  - Wildcards. extends and super keywords

Generic arguments in methods 1/2

- The Java compiler also erases type parameters in generic method arguments

```
1. public static <T> int count(T[] anArray, T elem) {
2.     int cnt = 0;
3.     for (T e : anArray)
4.         if (e.equals(elem))
5.             ++cnt;
6.     return cnt;
7. }
```

```
76: ldc           #3                  // String yes
78: invokestatic  #12                 // Method count:([Ljava/lang/Object;Ljava/lang/Object;)I
```

Generic arguments in methods 2/2

- Because T is unbounded, the Java compiler replaces it with Object:

```
1. public static int count(Object[] anArray, Object elem) {
2.     int cnt = 0;
3.     for (Object e : anArray)
4.         if (e.equals(elem))
5.             ++cnt;
6.     return cnt;
7. }
```

```
76: ldc           #3                  // String yes
78: invokestatic  #12                 // Method count:([Ljava/lang/Object;Ljava/lang/Object;)I
```

Generic arguments in methods

- Bounded types Java compiler replaces it with bounds

```
public static <T extends Comparable<T>> int
                                    countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray) {
        if (e.compareTo(elem) > 0) {
            ++count;
        }
    }
    return count;
}
```

```
57: invokestatic  #4                  // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
60: invokestatic  #10                 // Method countGreaterThan:([Ljava/lang/Comparable;Ljava/lang/Comparable;)I
```

Module contents

- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Wildcards. extends and super keywords

Wildcards. extends and super keywords 1/3

- The unbounded wildcard type is specified using the wildcard character (?)
- Because ? is Object, we can override the generic instance of the class to the same class instance with any type arguments

```
1. Pair<?, ?> p2 = new OrderedPair<Integer, Character>(12, 'k');
2. p2 = new OrderedPair<String, String>("aaa", "bbb");
3. p2 = new OrderedPair<int[], Long>(new int[]{1,2,3}, 100L);
```

Unbounded generic wildcard type arguments are useful:

- In the implementation of methods using the functionality of the Object class (not so much methods)
- When using methods that are independent of the type parameter

Wilcards are not allowed to be on the right side of an assignment

Unbounded Wildcard Type

```
public static void printList(List<Object> list) {
    for (Object x : list) {
        System.out.println(x);
    }
}

public static void main(String[] args) {
    List<String> keywords = new ArrayList<>();
    keywords.add("java");
    printList(keywords);       //incompatible types: List<String> cannot
                               // be converted to List<Object>
```

Unbounded Wildcard Type

```
public static void printList(List<?> list) {
   for (Object x : list) {
      System.out.println(x);
   }
}

public static void main(String[] args) {
   List<String> keywords = new ArrayList<>();
   keywords.add("java");
   printObjList(keywords);    //OK, List<String> match to any type list
```

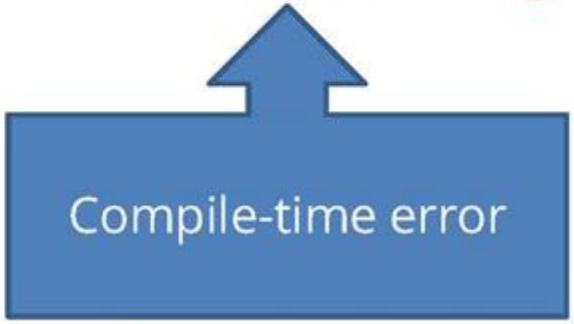
takes any type of list as a parameter

You cannot use metacharacters, either unlimited or limited, as type arguments.

Wildcards. extends and super keywords 2/3

- Bounded wildcards define what types can be used in a wildcard and what corresponding methods we can use
- Upper Bounded Wildcards
- Pair<? **extends** Number, ? > p2
  = **new** OrderedPair<Integer, Character>(12, **'k'**);
  p2 = **new** OrderedPair<Number, String>(99, **"bbb"**);
  p2 = **new** OrderedPair<Double, String>(9.99, **"bbb"**);

  p2 = **new** OrderedPair<**int**[], Long>(**new int**[]{1,2}, 100L);



Upper Bounded Wildcard Type

```
public static double sumNumberA(List<? extends Number> list) {
    double sum = 0;
    for (int i = 0; i < list.size(); i++) {
        sum += list.get(i).doubleValue();
```

takes Number or any of it descendant

```
public static double sumNumberA(java.util.List<? extends java.lang.Number>);
  Code:
    17: invokeinterface #57,  2           // InterfaceMethod java/util/List.get:(I)Ljava/lang/Object;
    22: checkcast     #61                 // class java/lang/Number
    25: invokevirtual #63                 // Method java/lang/Number.doubleValue:()D
}
```

```
public static <T extends Number> double sumNumberB(List<T> list) {
    double sum = 0;
    for (T n : list) {
        sum += n.doubleValue();
    }
```

method use the type argument

```
public static <T extends java.lang.Number> double sumNumberB(java.util.List<T>);
  Code:
    19: invokeinterface #77,  1           // InterfaceMethod java/util/Iterator.next:()Ljava/lang/Object;
    24: checkcast     #61                 // class java/lang/Number
    27: astore        4
...
```

Upper Bounded Wildcard Type

```
...
public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(1, 3, 5);
    List<Double> doubles = Arrays.asList(2.2, 4.4, 6.6);

    System.out.println(sumNumberA(ints));
    System.out.println(sumNumberA(doubles));

    System.out.println(sumNumberB(ints));
    System.out.println(sumNumberB(doubles));
}
```

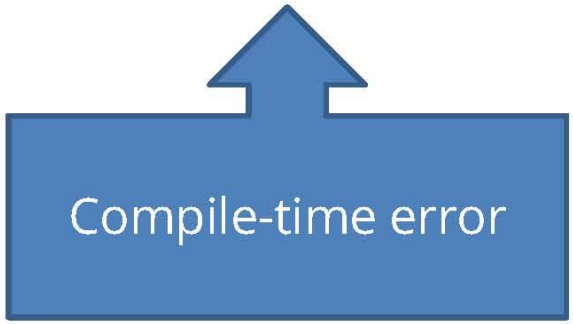
Output:

9.0
13.2
9.0
13.2

Wildcards. extends and super keywords 3/3

- Lower Bounded Wildcards

```
1. Pair<? super Integer, ? > p2
2.         = new OrderedPair<Integer, Character>(12, 'k');
3. p2 = new OrderedPair<Number, String>(99, "bbb");
4. p2 = new OrderedPair<Object, String>(99, "bbb");
5. p2 = new OrderedPair<int[], Long>(new int[]{1,2,3}, 100L);
```



Lower Bounded Wildcard Type

```
public static void addIntegers(List<? super Integer> list, Object obj) {
    list.add((Integer) obj);
}
```

method does not use the type argument

takes Integer or Number or Object as a type argument

```
public static <T super Integer> void addIntegersAgain(List<T> list,
                                                      Object obj) {
    list.add((T) obj);
}
...
```

Compile Error!!!

method use the type argument

We can not use lower bounded regular type variables!!!

Lower Bounded Wildcard Type

```
public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();
    List<Number> numList = new ArrayList<>();
    List objList = new ArrayList();
    int a = 5;
    addIntegers(intList, a);
    addIntegers(numList, a);
    addIntegers(objList, a);
    Number num = 20;
    addIntegers(intList, num);
    addIntegers(numList, num);
    addIntegers(objList, num);
    Object obj = 30;
    addIntegers(intList, obj);
    addIntegers(numList, obj);
    addIntegers(objList, obj);
...
...
    System.out.println(intList);
    System.out.println(numList);
    System.out.println(objList);
}
```

Output:

[5, 20, 30]
[5, 20, 30]
[5, 20, 30]