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> { /* ... */ }

- 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 MyClass<T> {...}

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

T, 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

See MyObjBox

See Main

See MyBox

See Main

Declaring and using generic types

- 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

- A generic class or interface can have multiple type parameters

See Pair

See OrderedPair

See Main

- JDK 14 records accept generics

See GenericRecord

See Main

- Generic use simplifies code

See comparator package

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

See GenericsMeth

generic method - generic algorithm!

- We can use different argument types for class and method with the same name parameter type

See MyTrickyBox

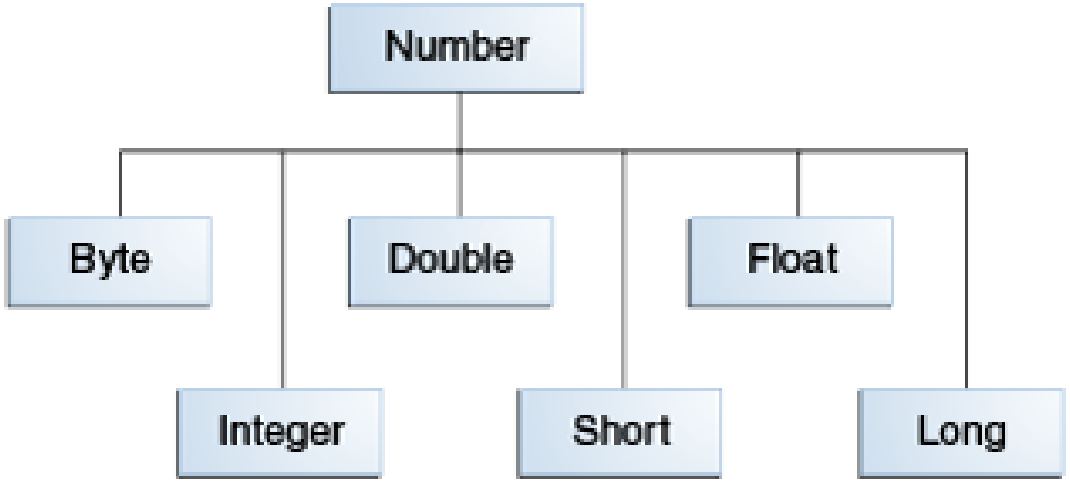
Module contents

- 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

Upper bound type parameter

```
// T must by Number or its subtype
public class MyBox<T extends Number> {...}
```



See MyBoundedBox

See Main

Bounded type parameters in generic methods

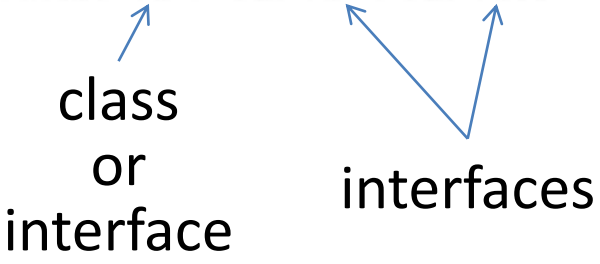
- Upper bounded type parameters in generic method are declared before method's return type

```
public <T extends Number> int countSum(T[] arr) {...}
```

See BoundedTypeMethod

Bounded Type Parameters

- 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

- 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.

See type.MyBox and type.Main by javap -c

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

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

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

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

See bounded.MyBoundedBox and bounded.Main by javap -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

Restrictions on Generics

1. Type arguments cannot be primitive types
2. You cannot create instances of type parameters
3. You cannot cast to or use instanceof types with type parameters
4. You cannot declare static fields whose types are type parameters
5. You cannot create arrays of parameterized types (but You can declare such arrays in generic classes and generic methods)
6. You cannot overload generic methods with the same methods with different type arguments
7. You cannot implement the same generic interfaces with different type arguments

See GenericsRestrictions

most restrictions - due to type erasure

Restrictions on Generics

8. Generic class cannot extend Throwable. 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)

See GenericsException

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

- The Java compiler also erases type parameters in generic method arguments
- Unbounded arguments type parameters the Java compiler replaces with Object
- Bounded arguments type parameters the Java compiler replaces with bounds

See mehod.GenericMeth by javap -c

See bounded.BoundedTypeMethod by javap -c

Module contents

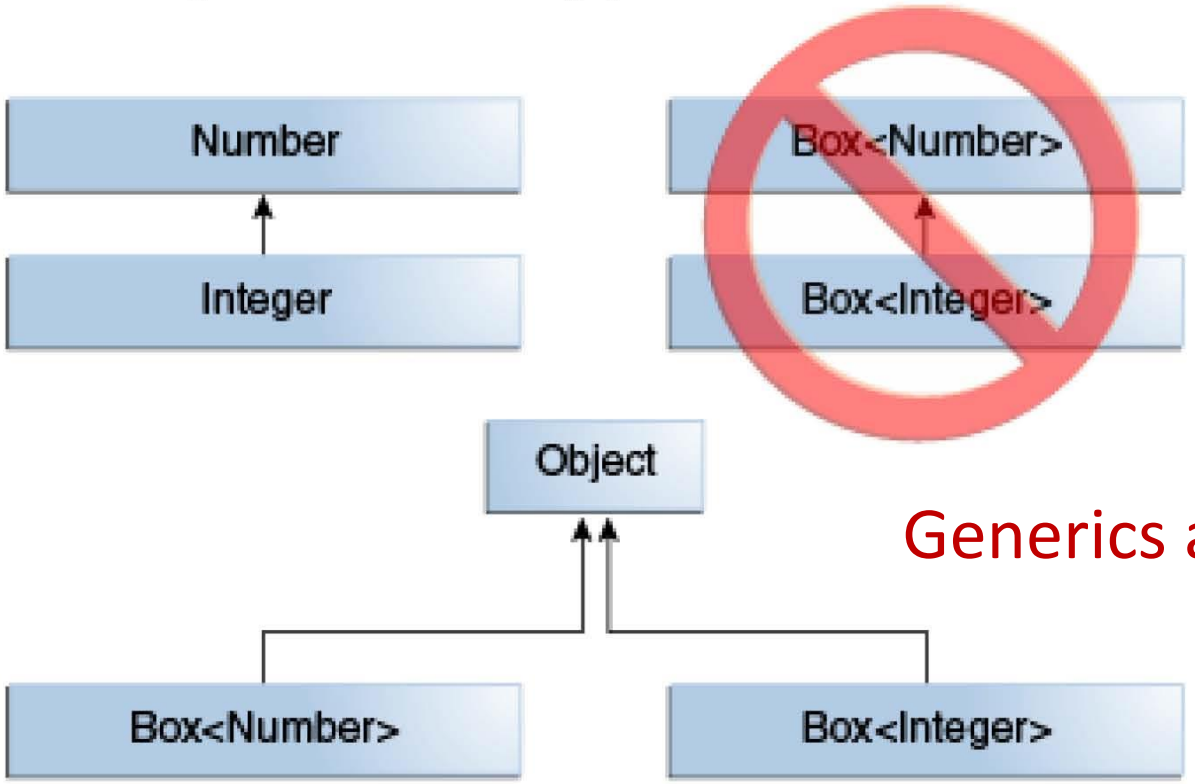
- Generics
  - The Generics
  - Declaring and using generic types
  - Declaring and using generic methods
  - Generics and JVM
  - Restrictions on Generics
  - Generic and inheritance
  - Generic arguments in methods
  - Covariance problemm
  - Wildcards. extends and super keywords

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 are invariant

See GenericsInherit

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

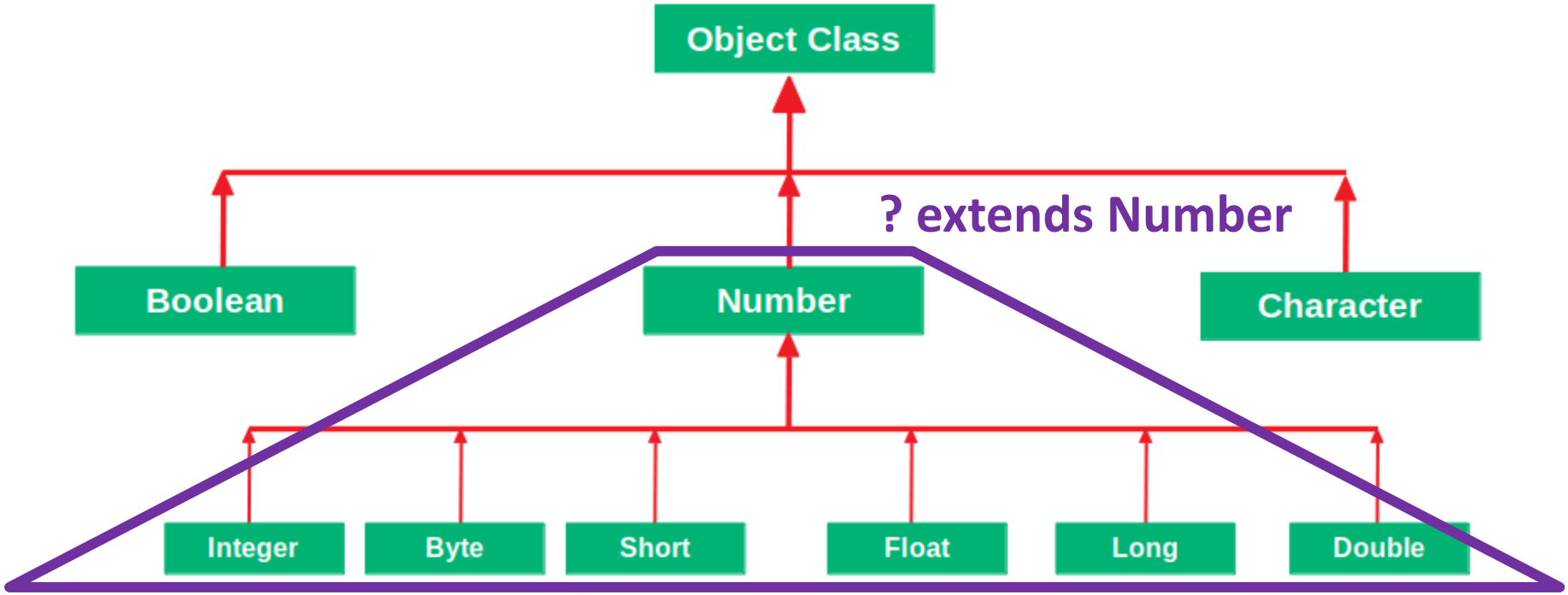
- The unbounded wildcard type is specified using the wildcard character (?)

See GenericsInherit

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

Wildcards. extends and super keywords

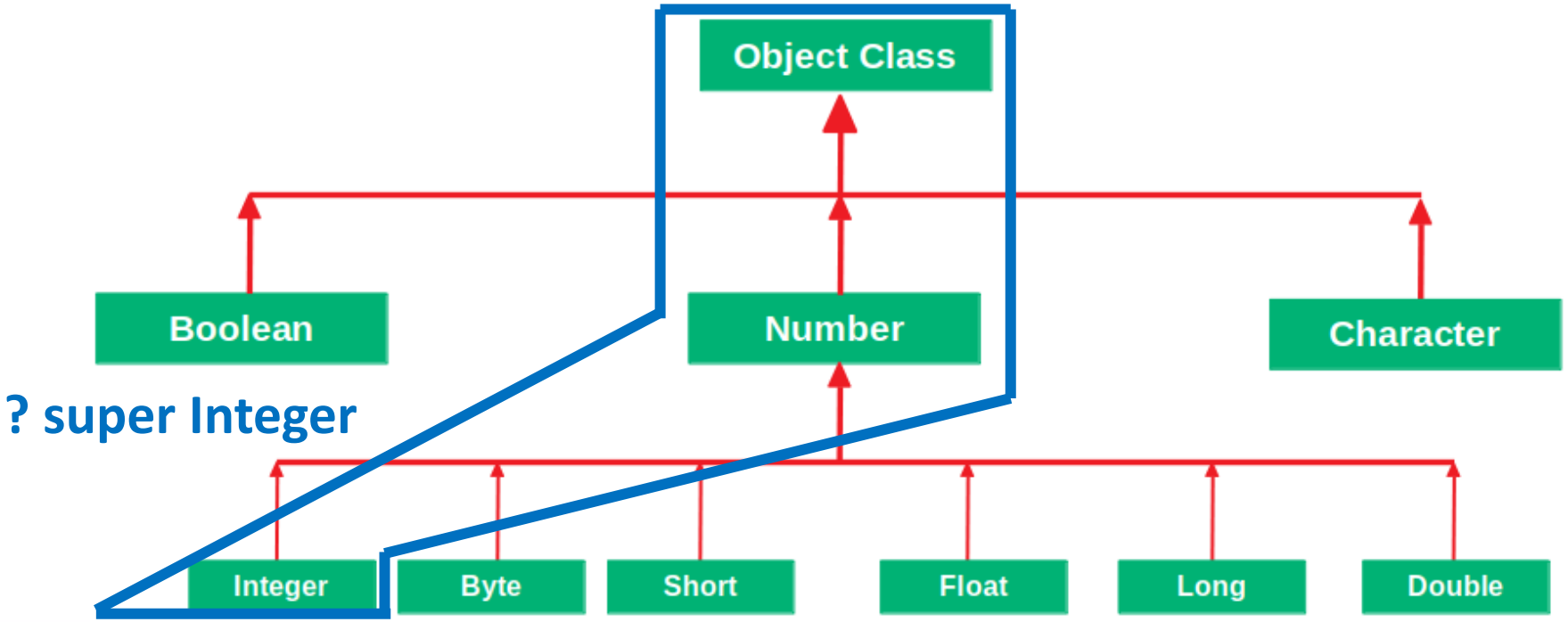
- Bounded wildcards define what types can be used in a wildcard and what corresponding methods we can use
- Upper bounded wildcards



See BoundedWildcardArgs

Wildcards. extends and super keywords

- Bounded wildcards define what types can be used in a wildcard and what corresponding methods we can use
- Lower bounded wildcards



See BoundedWildcardArgs

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
```

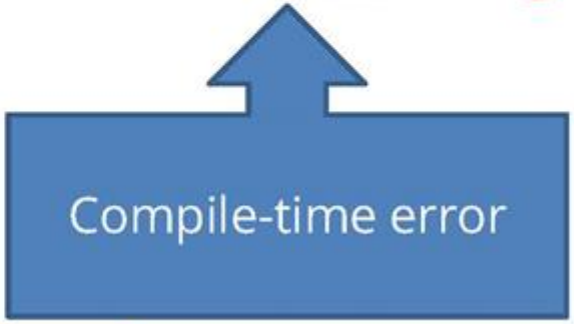
takes any type of list as a parameter

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));
}
```

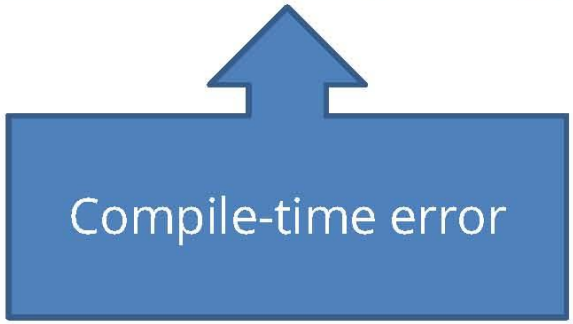
Output:

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]