PROGRAMMING III JAVA LANGUAGE

COURSE 5

PREVIOUS COURSE CONTENT

- **Generics**
  - Defining a generic
  - Run-time behavior

- **Collections**
  - List
  - Set
  - Map

COURSE CONTENT

- **Collections**
  - Utilities classes
- **Comparing objects**
- **Lambda expressions**
- **Generics**
  - Wild Cards
  - Restrictions

GENERICS

- **Introducesd in Java 1.5**

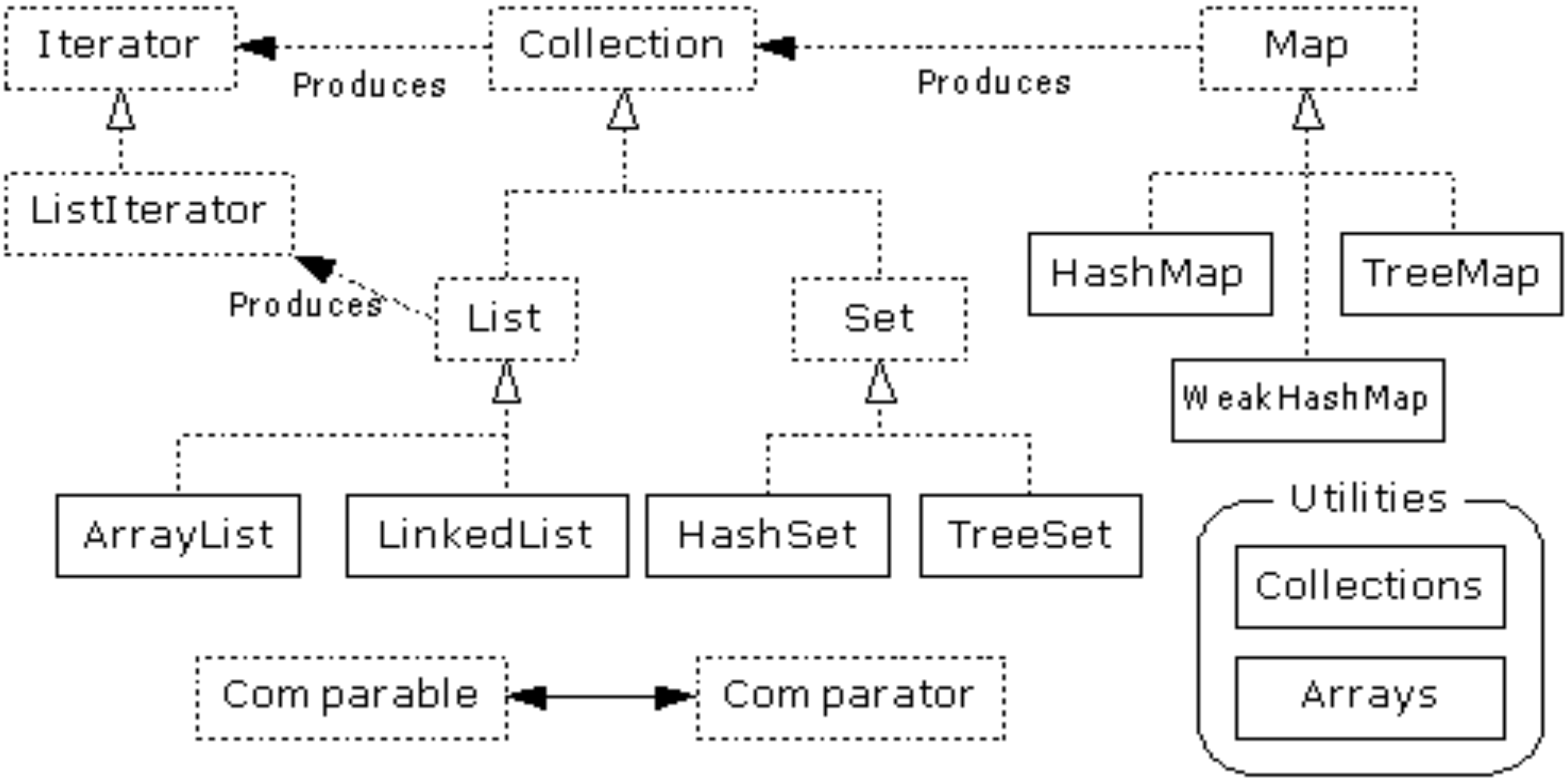
- **Allows class and methods definitions with parameters for types**
  - Classes or methods that have type parameters are called *parameterized class* or *generic definitions*, or, simply, *generics*

- **Can be defined by**
  - Java libraries
  - User

COLLECTIONS

- **What is a collection in Java?**
  - Containers of Objects which by polymorphism can hold any class that derives from Object
  - GENERICS make containers aware of the type of objects they store
    - from Java 1.5

COLLECTIONS. CLASS IERRHY



COLLECTIONS. OTHER CLASSES

- **Don't use for new development**
- **Still available for legasy**
  - `Hashtable`
    - use `HashMap`
  - `Enumeration`
    - use Collections and `Iterators`
  - `Vector`
    - use `ArrayList`
  - `Stack`
    - use `LinkedList`
  - `BitSet`
    - use `ArrayList` of `boolean`, unless you can't stand the thought of the wasted space
  - `Properties`

PROPERTIES CLASS

- **Located in java.util package**
- **Special case of Hashtable**
  - Keys and values are Strings
  - Tables can be saved to/loaded from file

- **Java VM maintains set of properties that define `system environment`**
  - Set when VM is initialized
  - Includes information about current user, VM version, Java environment, and OS configuration
  - Example:

```
Properties prop = System.getProperties();
Enumeration e = prop.propertyNames();
while (e.hasMoreElements()) {
        String key = (String) e.nextElement();
        System.out.println(key + " value is " +
                                            prop.getProperty(key));
}
```

COLLECTIONS IMPLEMENTATIONS

- **Creating special-case collections**
  - How?
    - Using decorator design pattern
  - Way?
    - added functionality, keep the Collections Framework simple,
  - Types
    - Read-only collections
    - Thread-safe collections
    - Singleton collections
    - Multiple copy collections
    - Empty collections

For more details see: http://pages.di.unipi.it/corradini/Didattica/PR2-B-14/Java%20Collections%20Framework.pdf

COLLECTIONS IMPLEMENTATIONS

- **Read-only collections**
  - added all the necessary elements to a collection, it may be convenient to treat that collection as read-only, to prevent the accidental modification of the collection
  - `unmodifiableCollection(), unmodifiableList(), unmodifiableMap(), unmodifiableSet(), unmodifiableSortedMap(), unmodifiableSortedSet ()`

- **Example**

```
public class ReadOnlyExample {
  public static void main(String args[]) {
   Set set = new HashSet();
   set.add("Bernadine");
   set.add("Elizabeth");
   set.add("Gene");
   set.add("Elizabeth");
   set = Collections.unmodifiableSet(set);
   set.add("Clara");
  }
}
```

UnsupportedOperationException is thrown

COLLECTIONS IMPLEMENTATIONS

- **Thread-safe collections**
  - new classes of Collections framework are *not* thread-safe
    - using a collection in a multi-threaded environment, where multiple threads can modify the collection simultaneously, the modifications need to be synchronized
  - `synchronizedCollection (), synchronizedList (), synchronizedMap (), synchronizedSet (), synchronizedSortedMap (), synchronizedSortedSet ()`

- **Example**

```
Set set = Collection.synchronizedSet(new HashSet());
```

COLLECTIONS IMPLEMENTATIONS

- **Singleton collections**
  - create single element sets

- **Example**

```
Set set = Collection.singleton("Hello");
```

COLLECTIONS IMPLEMENTATIONS

- **Multiple copy collections**
  - immutable list with multiple copies of the same element

- **Example**

```
List fullOfNullList =
        Collection.nCopies(10, null);
```

COLLECTIONS IMPLEMENTATIONS

- **Empty collections**
  - constants for empty collections
    - `List EMPTY_LIST`
    - `Set EMPTY_SET`
    - `Map EMPTY_MAP`

COLLECTIONS INITIALIZATION

- **Arrays**
  - Arrays.asList()
  - Example

```
List<String> fixedLengthList =
        Arrays.asList("C", "C++", "Java");
```

- **Collections**
  - Example

```
List<String> list = Collections.EMPTY_LIST;
Collections.addAll(list = new ArrayList<String>(), "C",
"C++", "Java");
```

- **Double Braces**
  - Example

```
List<String> list = new ArrayList<String>() {{
        add("C"); add("C++"); add("Java");
}};
```

- **Java 9 – Stream API**

STATIC METHODS FOR CREATING COLLECTIONS

```
List.of()
List.of(e1)
List.of(e1, e2) //fixed-arg overloads up to ten elements
List.of(elements...) //varargs supports arbitrary number of elements

Set.of()
Set.of(e1)
Set.of(e1, e2) //fixed-arg overloads up to ten elements
Set.of(elements...) //varargs supports arbitrary number of elements

Map.of()
Map.of(k1, v1)
Map.of(k1, v1, k2, v2) //fixed-arg overloads up to ten key-value pairs
Map.ofEntries(entry(k1, v1),entry(k2, v2),...) //varargs
Map.entry(k, v) //creates a Map.Entry instance
```

STATIC METHODS FOR CREATING COLLECTIONS

JAVA < 9

```
List<String> stringList =
       Arrays.asList("a", "b", "c");


Set<String> stringSet =
       new HashSet<>(Arrays.asList(
                       "a", "b", "c"));


Map<String, Integer> stringMap =
                     new HashMap<>();
stringMap.put("a", 1);
stringMap.put("b", 2);
stringMap.put("c", 3);
```

JAVA 9

```
List<String>stringList =
        List.of("a", "b", "c");


Set<String> stringSet =
        Set.of("a", "b", "c");


Map<String, Integer> stringMap =
       Map.of("a", 1,
              "b", 2,
              "c", 3);
```

IMMUTABILITY

- **Collections returned by the new static factory methods are immutable**
- **"Conventional" immutability, not "immutable persistent"**
  - atempts to `add()`, `set()`, or `remove()` throw `UnsupportedOperationException`
- **Immutability is good!**
  - Common case: collection initialized from known values, are never changed
  - Automatically thread-safe
  - Provides opportunities for efficiency, especially space

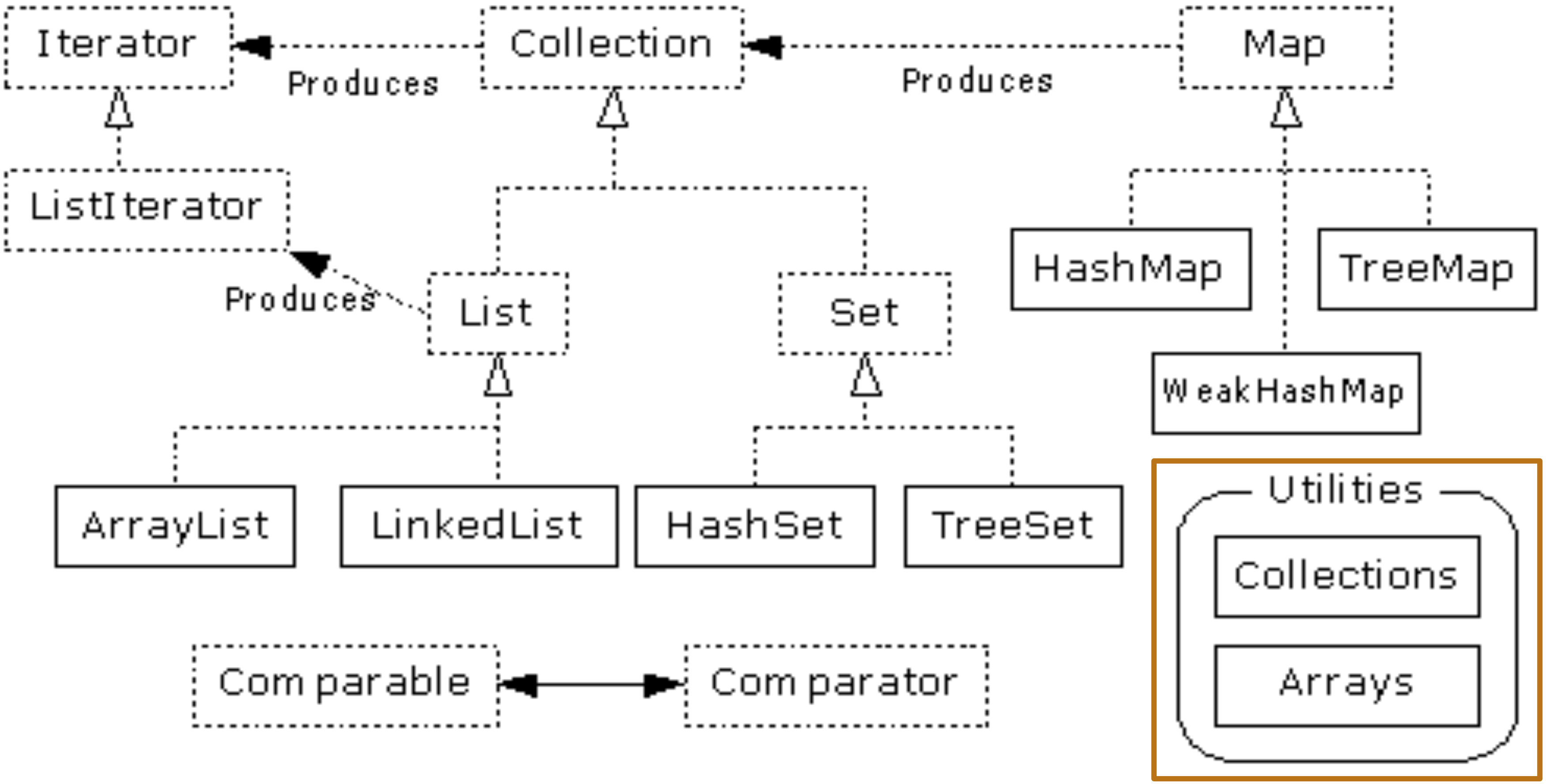
NULLS DISALLOWED

- **Nulls disallowed as `List` or `Set` members, `Map` keys or values**
  - `NullPointerException` thrown at creation time

- **Allowing nulls in collections back in 1.2**
  - no collection in Java 5 or later has permited nulls
  - particularly the `java.util.concurrent` collections

- **Why not?**
  - nulls are bad! source of NPEs
  - nulls useful as sentinel values in APIs, e.g., `Map.get()`
  - nulls useful as sentinel values for optimizing implementations

COLLECTIONS



COLLECTION. UTILITIES CLASS

- **Algorithms**
  - These are the methods that perform useful computations, such as *searching* and *sorting*, on objects that implement collection interfaces.
  - The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

- **The Collections class provides a number of static methods for fundamental algorithms**

- **Most operate on `Lists`, some on all `Collections`**
  - sort(), search(), shuffle()
  - reverse(), fill(), copy()
  - min(), max()

COMPARABLE AND COMPARATORS

- **Some classes provide the ability to sort elements.**
  - How is this possible when the collection is supposed to be de-coupled from the data?

- **Java defines two ways of comparing objects**
  - The objects implement the `Comparable` interface
  - A `Comparator` object is used to compare the two objects

- **If the objects in question are Comparable, they are said to be sorted by their "natural" order.**

- `Comparable` **object can only offer one form of sorting. To provide multiple forms of sorting,** `Comparator` **must be used.**

COMPARABLE INTERFACE

- **The Comparable interface contains the `compareTo()` method.**
  - `int compareTo( T obj)`

- **This method returns**
  - 0 if the objects are equal
  - <0 if this object is less than the specified object
  - >0 if this object is greater than the specified object.

- **In order to provide a natural ordering for objects, you must implement the `Comparable` interface**

- **Any object which is "Comparable" can be compared to another object of the same type.**
  - There is only one method defined within this interface.
  - Therefore, there is only one natural ordering of objects of a given type/class.

COMPARATOR INTERFACE

- **The Comparator interface defines two methods:**
  - `int compare(T obj1, T obj2)`
    - 0 if the Objects are equal
    - <0 if the first object is less than the second object
    - >0 if the first object is greater than the second object.
  - `boolean equals(T obj)`
    - returns true if the specified object is equal to this comparator (ie. the specified object provides the same type of comparison that this object does)

COMPARABLE AND COMPARATORS

- **Comparators are useful when objects must be sorted in different ways.**

- **For example**
  - Employees need to be sorted by first name, last name, start date, termination date and salary.
  - A comparator could be provided for each case
  - The comparator interrogates the objects for the required values and returns the appropriate integer based on those values.

- **The appropriate comparator is provided a parameter to the sorting algorithm.**

EXAMPLE JAVA 1.7

// two-level sort:

// sort by last name, then by nullable first name, nulls first

```
Collections.sort(students, new Comparator<Student>() {
  @Override
  public int compare(Student s1, Student s2) {
        int r = s1.getLastName().compareTo(s2.getLastName());

        if (r != 0)  return r;
        String f1 = s1.getFirstName();
        String f2 = s2.getFirstName();

        if (f1 == null) {
                return f2 == null ? 0 : -1;
        } else {
                return f2 == null ? 1 : f1.compareTo(f2);
        }
  }
});
```

The array of objects to be sorted

Aninomus inner class implementing `Comparator` interface

EXAMPLE JAVA 1.8

// two-level sort:

// sort by last name, then by nullable first name, nulls first

The array of objects to be sorted

```
Collections.sort(students, (s1, s2) ->  {
          int r = s1.getLastName().compareTo(s2.getLastName());

          if (r != 0)  return r;
          String f1 = s1.getFirstName();
          String f2 = s2.getFirstName();
          if (f1 == null) {
                  return f2 == null ? 0 : -1;
          } else {
                  return f2 == null ? 1 : f1.compareTo(f2);
          }
    }
});
```

Lambda expression for implementing `Comparator` interface

LAMBDA EXPRESSIONS

- **A Java 8 lambda is basically a method in Java without a declaration usually written as** `(parameters) -> { body }`.
  - Examples
    - `(int x, int y) -> { return x + y; }`
    - `x -> x * x`
    - `( ) -> x`

- **A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context.**
  - Parenthesis are not needed around a single parameter.
  - ( ) is used to denote zero parameters.

- **The body can contain zero or more statements.**
  - Braces are not needed around a single-statement body.

LAMBDA EXPRESSIONS

- **Example of lambda usage for iterating through a list**
  - `List<Integer> intSeq = Arrays.asList(1, 2, 3);`
  - `intSeq.forEach(z -> System.out.println(z));`
- *`x -> System.out.println(x)`* **is a lambda expression that defines an anonymous function with one parameter named** `x` **of type** `Integer`
- **How could lambda be used to iterate through a map**

```
Map<String, Integer> items = new HashMap<>();
items.put("A", 10);
items.put("B", 20);
items.forEach((k, v) -> System.out.println("key : "
+ k + " value : " + v));
```

FUNCTIONAL INTERFACES

- **Interfaces with only one explicit abstract method**
  - AKA SAM interface (Single Abstract Method)
- **Optionally annotated with @FunctionalInterface**
  - Do it, for the same reason you use @Override
- **Some Functional Interfaces you know**
  - `java.lang.Runnable`
  - `java.util.concurrent.Callable`
  - `java.util.Comparator`
  - `java.awt.event.ActionListener`
  - `Many, many more in package java.util.function`

METHOD REFERENCES

- **An alternative to lambda**
  - An instance method of a particular object (bound)
    - `objectRef::methodName`
  - An instance method, whose receiver is unspecified (unbound)
    - `ClassName::instanceMethodName` – The resulting function has an extra argument for the receiver
  - A static method
    - `ClassName::staticMethodName`
  - A constructor
    - `ClassName::new`

GENERICS. WILDCARDS

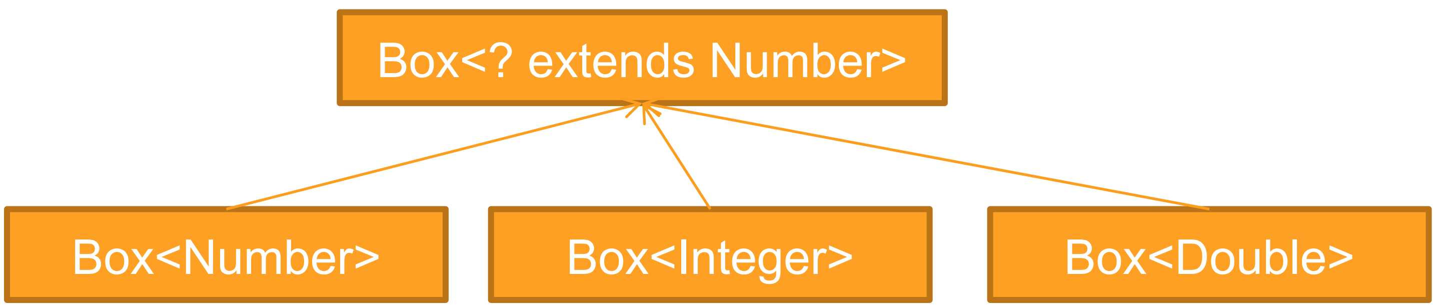
- **Bounded Type Parameters**
  - restrict the types that can be used as type arguments in a parameterized type
    - <T extends B1 [& B2 [& B3 …]]>
- **Wildcards**
  - Wildcard - `?`
    - Represents an unknown type
  - Can be used as the type of a
    - Parameter
    - Field
    - Local variable
    - Sometimes as a return type

GENERICS. WILDCARDS

- **Upper Bounded Wildcards**
  - `public static void process(List<? extends Foo> list)`
- **Unbounded Wildcards**
  - `public static void printList(List<?> list)`
- **Lower Bounded Wildcards**
  - `public static void addNumbers(List<? super Integer> list)`

GENERICS. WILDCARDS. UPPER BOUNDED

- **Upper Bounded**
  - Defines a type that is bounded by the superclass
  - Example
    - Create a class box that can contain only objects that are subtypes of class number




```
Box<? extends Number> box = new Box<Integer>()
```

GENERICS. WILDCARDS. UPPER BOUNDED

```
public class Box<E> {
   public void copyFrom(Box<E> b) {
         this.data = b.getData();
    }
}

Box<Integer> intBox = new Box<>();
Box<Number> numBox = new Box<Number>();
numBox.copyFrom(intBox);
```

- ❑ **Does the code execute?**

Error incompatible types `Number` and `Integer`

GENERICS. WILDCARDS. UPPER BOUNDED

```
public class Box<E> {
   public void copyFrom(Box<E extends Number> b) {
         this.data = b.getData();
    }
}

Box<Integer> intBox = new Box<>();
Box<Number> numBox = new Box<Number>();
numBox.copyFrom(intBox);
```

GENERICS. WILDCARDS. UNBOUNDED

- **Unbounded Wildcards**
  - Method to print any list of any type of objects

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

  - Call

```
List<Object> listObject = new ArrayList<>();
printList(listObject);

List<String> listString = new ArrayList<>();
printList(listString); //-> compilation error
```

  - How to resolve?

GENERICS. WILDCARDS. UNBOUNDED

- **Unbounded Wildcards**
  - Print any list of objects

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

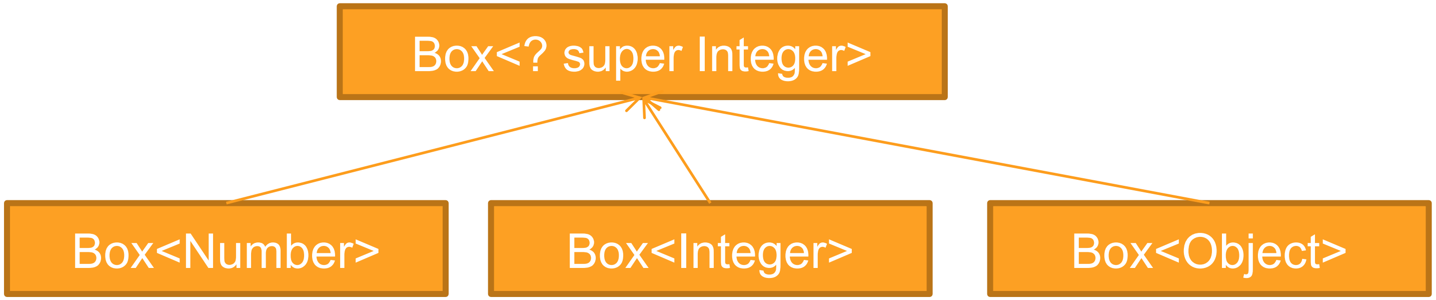
  - Call

```
List<Object> listObject = new ArrayList<>();
printList(listObject);

List<String> listString = new ArrayList<>();
printList(listString);
```

GENERICS. WILDCARDS. LOWER BOUNDED

- **Lower Bounded**
  - Defines a type that is bounded by the subclass
  - Example
    - Create a class box that can contain only objects that are subtypes of class number



- `Box<? super Integer> box = new Box<Number>()`

GENERICS. WILDCARDS. LOWER BOUNDED

- **Suppose we want to write** `copyTo()` **that copies data in the opposite direction of** `copyFrom()`**.**
  - `copyTo()` copies data from the host object to the given object.

```
public void copyTo(Box <E>b) {
   b.data = this.getData();
}
```

- **Above code is fine as long as** `b` **and the host are boxes of exactly same type. But** `b` **could be a box of an object that is a superclass of** `E`**.**

```
public void copyTo(Box<? super E> b) {
    b.data = this.getData();
}

 Box <Integer> intBox = new Box<>();
 Box <Number> numBox = new Box<>();
 intBox.copyTo(numBox);
```

GENERICS. RESTRICTIONS

- **Java, generic types are compile-time entities**
  - C++, instantiations of a class template are compiled separately as source code, and tailored code is produced for each one
  - Primitive type parameters (`List<int>`) not allowed
    - in C++, both classes and primitive types allowed
    - Java – auto boxing
  - Objects in JVM have non-generic classes

```
Pair<String> strPair = new Pair <>();
Pair<Number> numPair = new Pair<>().;
b = strPair.getClass () == numPair.getClass ();
assert b == true; // both of the raw class Pair
```

  - But byte-code has reflective info about generics

GENERICS. RESTRICTIONS

- **Instantiations of generic parameter T are not allowed**
  - `new T () // ERROR: whatever T to produce?`
  - `new T [10]`

- **Arrays of parameterized types are not allowed**
  - `new Pair<String> [10]; // ERROR`
  - since type erasure removes type information needed for checks of array assignments

- **Static fields and static methods with type parameters are not allowed**
  - `private static T singleOne; // ERROR`
  - since after type erasure, one class and one shared static field for all instantiations and their objects

- **Cannot Create, Catch, or Throw Objects of Parameterized Types**

GENERICS

- **Why generic programming**
  - supports *statically-typed* data structures
    - *early detection* of type violations
      - cannot insert a string into ArrayList <Number>
    - also, hides automatically generated casts
  - *superficially* resembles C++ templates
    - C++ templates are factories for ordinary classes and functions
      - a new class is always instantiated for given distinct generic parameters (type or other)
  - generic types are factories for *compile-time* entities related to types and methods