



# Object Oriented Programming with Java

08 – Generics & Collections

# What are generics?

- Allows to specify a generic implementation for class, interface or method
- Generic parameters can be defined at compile time using angle brackets (<>)
- Used mostly in collections

# Defining generics

- Defining a generic class

```
class Rental<T>{  
    T item;  
}
```

- Defining generic interfaces

```
interface List<E>{}
```

- Defining generic methods

```
public <U> void rentout(U item) {...}
```

# Calling generics

- Regarding the definitions on the previous slide:

```
Rental<Car> rentalCar = new Rental<Car>();  
Rental<House> rentalHouse = new Rental<House>();  
  
Car subaru = new Car();  
rentalCar.rentout(subaru);  
  
House myHouse = new House();  
rentalHouse.rentout(myHouse);
```

# Constraining Generic Definition

- Generics can be defined as to be extended from a certain class or interface

```
class Rental<T extends Vehicle>{  
    T item;  
}
```

# Wildcards

- We use wildcards to broaden usage of generic types. Code below only accepts items of type Object, no subtypes

```
public void setRental(Rental<Object> rental) {...}
```

- By using <?> wildcard we can extend the scale of the parameter

```
public void setRental(Rental<?> rental) {...}
```

- We can also add a constraint to the wildcard

```
public void setRental(Rental<? extends Vehicle> rental) {...}  
//or  
public void setRental(Rental<? super Car> {...})
```

# WildCards with Collections

- If we define a method taking a collection as parameter like below:

```
public void addAnimal(ArrayList<Animal> animals){  
    animals.add(new Dog());  
}
```

- If we pass an ArrayList<Dog> into the method we get a compiler error. We can do it as shown below:

```
public void addAnimal(ArrayList<? extends Animal> animals){  
    animals.add(new Dog());  
}
```

# Generic Methods

- We have seen the first usage of generic methods → using a generic class
- The second usage is defining a generic method free of a generic class:

```
public <C1 extends Car, C2 extends C1> void buyCar(C1 car, C2 oldCar)
{...}
```

- Usage:

```
//FamilyCar and SportsCar are subclasses of Car
Car myBrand = new Car();
FamilyCar volvo = new FamilyCar();
SportsCar porsche = new SportsCar();
Dealer carDealer = new Dealer();
//buy a new car trading the old one
carDealer.buyCar(myBrand, volvo);
```



# Case Study – Custom List

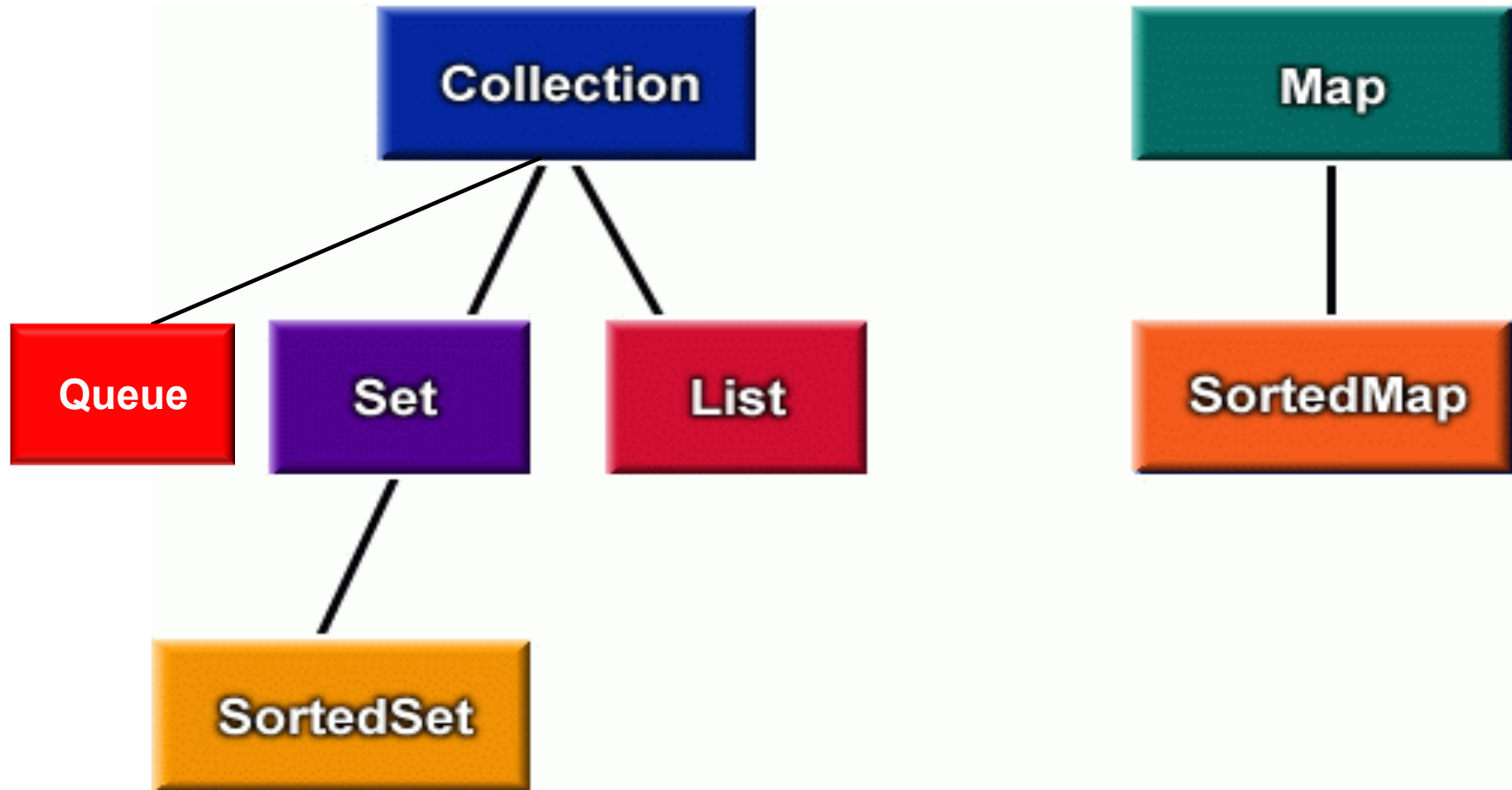
- Create your own custom list using the generic types
- Items can be added and returned from an index using the custom list
- For more please visit:

<http://download.oracle.com/javase/tutorial/java/generics/index.html>

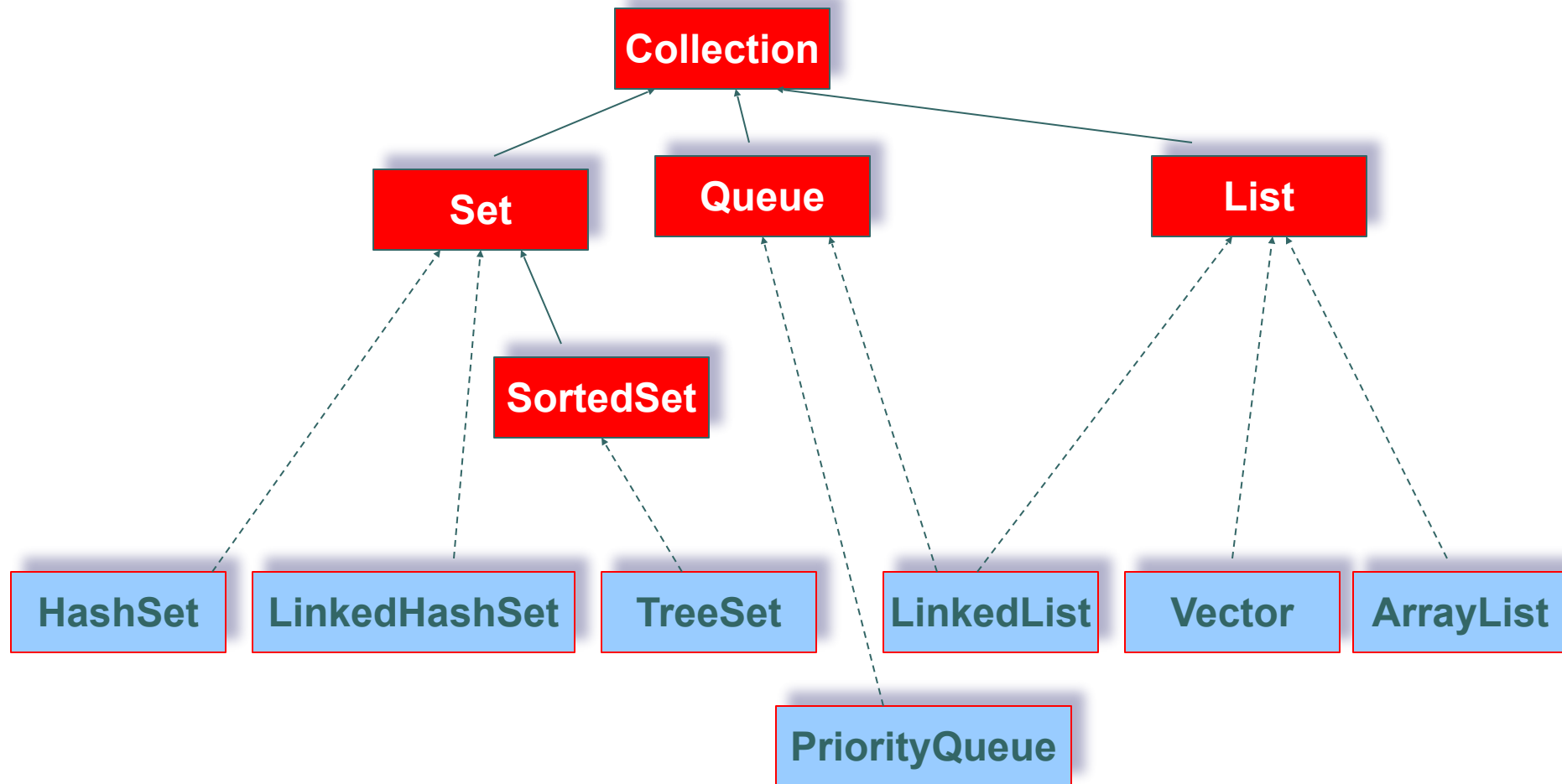
# Java Collections Framework

- The Collections Framework consists of interfaces and their implementations..
- A *collection* is a *container* or *object* that groups multiple objects into a single unit
- By implementing some interfaces by provided by Sun Microsystems you can develop your own collections

# Collections Interfaces



# Collection Implementation Classes



# Collection Interfaces

- **Collection** : Basic set of methods for working with data structures
  - No ordering
  - May contain duplicate elements
  - Has no direct implementation class
- **List** extends Collection :
  - Elements kept in inserted order
  - Any element can be accessed by using index no (using **get()** method)
  - Can be searched (using **contains()** method)
  - May contain **null** elements
  - Implementation classes are ;  
***LinkedList, Vector, ArrayList***

# Collection Interfaces

- **Set** extends Collection : A collection that has no duplicate elements
  - Can be searched (using **contains()** method)
  - No duplicate elements (no pair elements)
  - May contain one **null** element (at most)
  - Elements not kept in inserted order unless `LinkedHashSet` used
  - Implementation classes are ; **HashSet**, **LinkedHashSet**
- **SortedSet** extends Set :
  - Has all features of Set
  - Elements kept sorted
  - Implementation classes is; **TreeSet**

# Queue

- **Queue extends Collection :** A collection that supports ordering on FIFO basis
  - Has extra methods in addition to Collection interface's, Deque is an intermediate interface
  - Implementation: LinkedList (implements Queue), ArrayDeque (implements Deque)
    - add(e) → offer(e)
    - remove() → poll()
    - element() → peek()

# Concrete Classes from List

- **ArrayList** (Basic one, fast indexing)
- **Vector** (Synchronized slower than ArrayList)
- **LinkedList** (addFirst & addLast, slow indexing)



# Concrete Classes from Set

- **HashSet** (Basic one)
- **LinkedHashSet** (Guaranties the insertion order )
- **TreeSet** (extends SortedSet)
- Notes:
  - Sets get use of the hashCode() and equals() methods to catch duplicates, these methods can be overridden
  - For objects to be able to be sorted by the TreeSet, they should implement Comparable interface and override compareTo() method

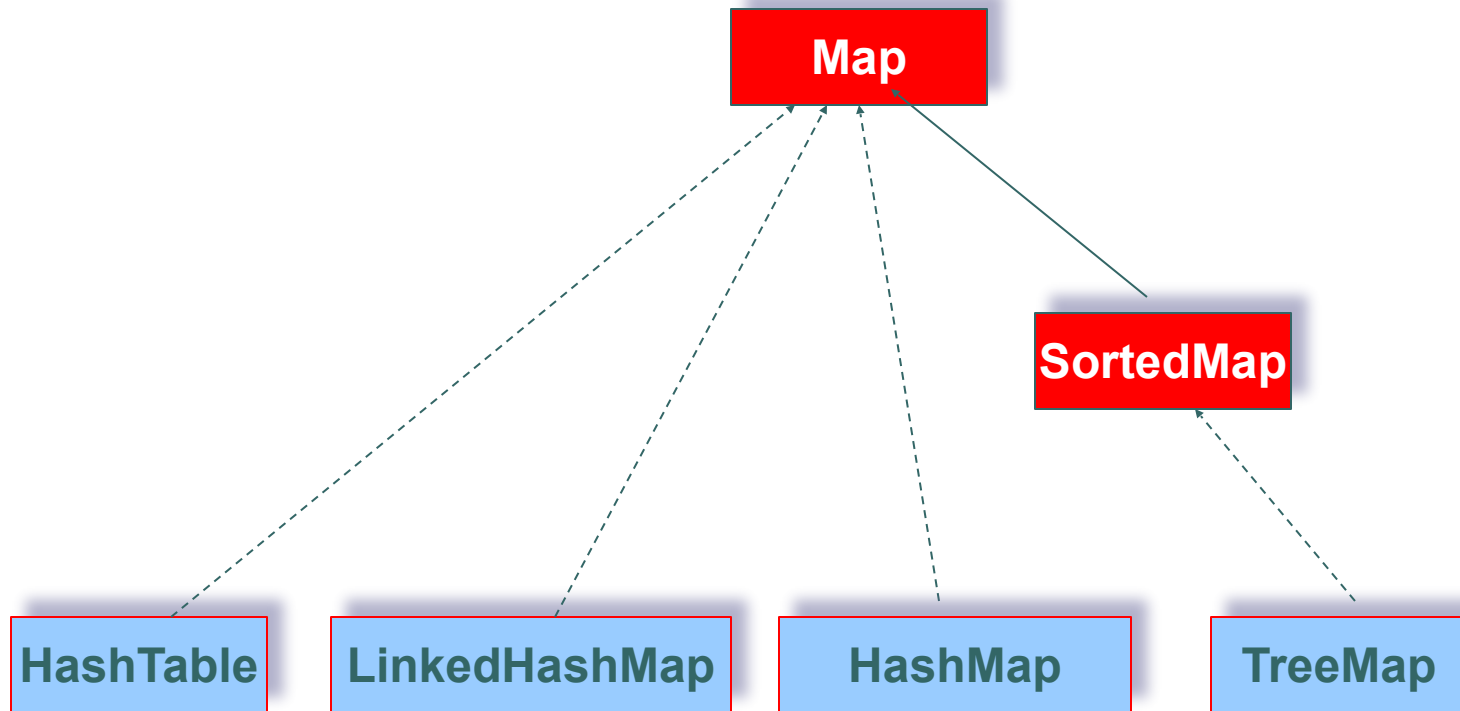
# hashCode() and Comparable Interface - Example

```
public class Student implements Comparable {
    int number;
    public Student(int no) {
        this.number = no;
    }
    @Override
    public int compareTo(Object o) {
        if(this.number==((Student)o).number){
            return 0;
        }else if(this.number>((Student)o).number){
            return 1;
        }else{
            return -1;
        }
    }
    @Override
    public int hashCode(){
        return this.number;
    }
    @Override
    public boolean equals(Object obj) {
        if(this.number == ((Student)obj).number){
            return true;
        }else{ return false;}
    }
}
```

for student objects to be sortable compareTo() method should be overridden  
 return 0 : compared object same  
 -1: compared object larger than me  
 1: compared object smaller than me

for non duplicates in sets both methods must be overridden, now students with the same number will be accepted as same objects

# Map Implementation Classes



# Map Interfaces

- **Map** does NOT extend Collection : An object that maps keys to values
  - No duplicate keys
  - Values can be accessed rapidly by using keys
  - Implementation classes are ;  
**HashTable, HashMap, LinkedHashMap**
- **SortedMap** extends Map :
  - Has all features of Map
  - Keys are kept sorted (natural order or Comparable interface)
  - Implementation class is ; **TreeMap**

# Map Concrete Classes

- **HashMap** ( Basic one may not be ordered)
- **LinkedHashMap** (Guaranties the insertion order)
- **HashTable** (*Synchronized* , do not allow null keys or values)
- **TreeMap** (extends SortedMap so keys are sorted by natural order)

# Map Example

- We use put() method in order to add elements to a map
- to iterate over a map, first we can get an iterator object from [map].keySet() which returns a set of keys or we can just use an enhanced for with the set of keys

```
public static void main(String[] args) {

    HashMap<Integer, String> myMap = new HashMap<Integer, String>();
    myMap.put(1322, "Ali");
    myMap.put(2332, "Ahmet");
    myMap.put(3442, "Mehmet");

    Set myKeySet = myMap.keySet();

    for (Iterator iterator = myKeySet.iterator(); iterator.hasNext();) {
        int key = (Integer)iterator.next();
        System.out.println(myMap.get(key));
    }
}
```

# Deque Interface

- A collection that can be used as a stack or a queue
  - It means a “double-ended queue” (and is pronounced “deck”).
  - A queue provides FIFO (first in, first out) operations: – add(e) and remove() methods
  - A stack provides LIFO (last in, first out) operations: – push(e) and pop() methods

# Stack with Deque: Example

```
1 public class TestStack {
2     public static void main(String[] args){
3         Deque<String> stack = new ArrayDeque<>();
4         stack.push("one");
5         stack.push("two");
6         stack.push("three");
7
8         int size = stack.size() - 1;
9         while(size>=0){
10            System.out.println(stack.pop());
11            size--;
12        }
13    }
14 }
```



# Arrays Class - Sorting an Array

- We can sort an array using Arrays class's sort() method

```
String[] isimler = {"Zeynep", "Mustafa", "Ahmet", "Vedat", "Ali"};

System.out.println(Arrays.toString(isimler));
Arrays.sort(isimler);

    System.out.println("After Array Sort");

    for(String isim : isimler)
    {
        System.out.println(isim);
    }
```

# Arrays Class - From Array to List

- We can convert an array to a list by using Arrays class's asList() method

```
String[] isimler = {"Zeynep", "Mustafa", "Ahmet", "Vedat", "Ali"};

List isimlist = Arrays.asList(isimler);

for (int i = 0; i < isimlist.size(); i++) {
    System.out.println((String)isimlist.get(i));
}
```

# Static Methods of Collections Class

- Consists of static methods that operate on or return collections
- can shuffle, sort, reverse... collections

```
String[] isimler = {"Zeynep", "Mustafa", "Ahmet", "Vedat", "Ali"};
List isimlist = Arrays.asList(isimler);
Collections.shuffle(isimlist);
for (int i = 0; i < isimlist.size(); i++) {
    System.out.println((String) isimlist.get(i));
}
```

## Case Study: Occurrences of Words

- Write a program that calculates the frequencies of words in a text and displays the words and their number of occurrence in an alphabetical order.