Generics, Collections, & Iterators

Lecture 5 (23 february 2021)

Introduction to Java Programming (Liang): chapters 19, 20, & 21

Radboud University

GENERICS

software development

goal

- correct, robust, adaptable and extendable
- fast development, reuse existing parts/libraries

software development

goal

- correct, robust, adaptable and extendable
- fast development, reuse existing parts/libraries

OO tools/principles

- **encapsulation**: information hiding
- **realization**: implements
- **composition**: has-a
- **inheritance**: is-a, extends
- **polymorphism**: overriding

software development

goal

- correct, robust, adaptable and extendable
- fast development, reuse existing parts/libraries

OO tools/principles

- **encapsulation**: information hiding
- **realization**: implements
- **composition**: has-a
- **inheritance**: is-a, extends
- **polymorphism**: overriding

Java additions

- **strong static typing**: spot all type problems at compile time

software development

goal

- correct, robust, adaptable and extendable
- fast development, reuse existing parts/libraries

OO tools/principles

- **encapsulation**: information hiding
- **realization**: implements
- **composition**: has-a
- **inheritance**: is-a, extends
- **polymorphism**: overriding

Java additions

- **strong static typing**: spot all type problems at compile time
- **exception mechanism** to handle runtime errors

software development

goal

- correct, robust, adaptable and extendable
- fast development, reuse existing parts/libraries

OO tools/principles

- **encapsulation**: information hiding
- **realization**: implements
- **composition**: has-a
- **inheritance**: is-a, extends
- **polymorphism**: overriding

Java additions

- **strong static typing**: spot all type problems at compile time
- **exception mechanism** to handle runtime errors
- **generics**: increase reusability and type safety by type parameters

the design of `equals`

`==` operator checks equality of object-locations (pointer comparison)

the design of `equals`

`==` operator checks equality of object-locations (pointer comparison)

the default Object **`equals`** method does the same:

```
public boolean equals (Object o) {
  return this == o;
}
```

the design of `equals`

`==` operator checks equality of object-locations (pointer comparison)

the default Object **`equals`** method does the same:

```
public boolean equals (Object o) {
  return this == o;
}
```

typically you want to check equality of (some) attributes

- override the **`equals`** for your class

the design of `equals`

`==` operator checks equality of object-locations (pointer comparison)

the default Object **`equals`** method does the same:

```
public boolean equals (Object o) {
  return this == o;
}
```

typically you want to check equality of (some) attributes

- override the **`equals`** for your class

desirable property: **symmetry**

- `x.equals(y)` ⇔ `y.equals(x)`

the design of `equals`

`==` operator checks equality of object-locations (pointer comparison)

the default Object **`equals`** method does the same:

```
public boolean equals (Object o) {
  return this == o;
}
```

`false` if there is no object or it has another type

typically you want to check equality of (some) attributes

- override the **`equals`** for your class

desirable property: **symmetry**

- `x.equals(y)` ⇔ `y.equals(x)`

the design of `equals`

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

the design of equals

```java
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```java
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

the design of equals

```java
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```java
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

or some other choice

the design of equals

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.equals(s) = " + p.equals(s));
  System.out.println("s.equals(p) = " + s.equals(p));
}
```

```
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

or some other choice

the design of equals

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.equals(s) = " + p.equals(s));
  System.out.println("s.equals(p) = " + s.equals(p));
}
```

```
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

or some other choice

```
p.equals(s) = false
s.equals(p) = false
```

the design of equals

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

or some other choice

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.equals(s) = " + p.equals(s));
  System.out.println("s.equals(p) = " + s.equals(p));
}
```

```
p.equals(s) = false
s.equals(p) = false
```

(Person) Student s would be equal to p

the design of `equals`: inheritance safe

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Person)) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

the design of `equals`: inheritance safe

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Person)) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Student)) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

the design of `equals`: inheritance safe

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Person)) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Student)) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.equals(s) = " + p.equals(s));
  System.out.println("s.equals(p) = " + s.equals(p));
}
```

the design of `equals`: inheritance safe

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Person)) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Student)) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.equals(s) = " + p.equals(s));
  System.out.println("s.equals(p) = " + s.equals(p));
}
```

```
p.equals(s) = true
s.equals(p) = false
```

the design of equals: inheritance safe

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Person)) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Student)) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.equals(s) = " + p.equals(s));
  System.out.println("s.equals(p) = " + s.equals(p));
}
```

```
p.equals(s) = true
s.equals(p) = false
```

violates x.equals(y) ⇔ y.equals(x)

the design of `equals`: inheritance safe

```
public class Person {
  protected String name;
  public Person(String name) {
    this.name = name;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Person)) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }
}
```

```
public class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Student)) {
      return false;
    }
    final Student other = (Student) obj;
    return this.num == other.num;
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.equals(s) = " + p.equals(s));
  System.out.println("s.equals(p) = " + s.equals(p));
}
```

```
p.equals(s) = true
s.equals(p) = false
```

might still be a good design choice

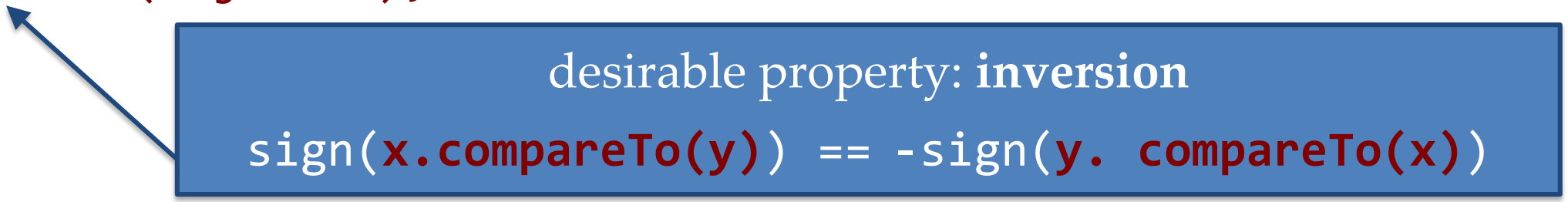
violates `x.equals(y)` ⇔ `y.equals(x)`

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```



design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {




  }
```

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {

    final Person q = (Person) obj;
    return name.compareTo(q.name);
  }
```

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {
    if (! (obj instanceof Person)) {

    }
    final Person q = (Person) obj;
    return name.compareTo(q.name);
  }
```

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {
    if (! (obj instanceof Person)) {
      return ??;
    }
    final Person q = (Person) obj;
    return name.compareTo(q.name);
  }
```

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {
    if (! (obj instanceof Person)) {
      return ??;
    }
    final Person q = (Person) obj;
    return name.compareTo(q.name);
  }
```

any integer is wrong!

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {
    if (! (obj instanceof Person)) {
      return ??;
    }
    final Person q = (Person) obj;
    return name.compareTo(q.name);
  }
```

any integer is wrong!

1) use another result type

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {
    if (! (obj instanceof Person)) {
      return ??;
    }
    final Person q = (Person) obj;
    return name.compareTo(q.name);
  }
```

any integer is wrong!

1) use another result type
2) throw an exception

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {
    if (! (obj instanceof Person)) {
      return ??;
    }
    final Person q = (Person) obj;
    return name.compareTo(q.name);
  }
```

any integer is wrong!

1) use another result type
2) throw an exception
3) avoid wrong type of argument

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {
    if (! (obj instanceof Person)) {
      return ??;
    }
    final Person q = (Person) obj;
    return name.compareTo(q.name);
  }
```

any integer is wrong!

1) use another result type
2) throw an exception
3) avoid wrong type of argument

design of compare

```
interface Comparable {
  int compareTo(Object o);
}
```

Note: this is not the definition of Comparable in the standard Java API

desirable property: **inversion**

`sign(x.compareTo(y)) == -sign(y. compareTo(x))`

for **Person**, in the style of **equals**:

```
  public int compareTo(Object obj) {
    if (! (obj instanceof Person)) {
      return ??;
    }
    final Person q = (Person) obj;
    return name.compareTo(q.name);
  }
```

any integer is wrong!

1) use another result type
2) throw an exception
3) avoid wrong type of argument

a type-safe compare

```
interface Comparable<T> {
  int compareTo(T o);
}
```

a type-safe compare

generic type argument:
"formal generic type"

```
interface Comparable<T> {
  int compareTo(T o);
}
```

a type-safe compare

generic type argument: "formal generic type"

using the formal generic type

```
interface Comparable<T> {
  int compareTo(T o);
}
```

a type-safe compare

generic type argument: "formal generic type"

using the formal generic type

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

a type-safe compare

generic type argument: "formal generic type"

using the formal generic type

generic instantiation: "actual concrete type"

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

a type-safe compare

generic type argument: "formal generic type"

using the formal generic type

generic instantiation: "actual concrete type"

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

typing: o is Person

a type-safe compare

generic type argument: "formal generic type"

using the formal generic type

generic instantiation: "actual concrete type"

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

typing: o is Person

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Student s) {
    return num - s.num;
  }
}
```

a type-safe compare

generic type argument: "formal generic type"

using the formal generic type

generic instantiation: "actual concrete type"

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

typing: o is Person

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Student s) {
    return num - s.num;
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.compareTo(s) = " + p.compareTo(s));
  System.out.println("s.compareTo(p) = " + s.compareTo(p));
}
```

a type-safe compare

generic type argument: "formal generic type"

using the formal generic type

generic instantiation: "actual concrete type"

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

typing: o is Person

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Student s) {
    return num - s.num;
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.compareTo(s) = " + p.compareTo(s));
  System.out.println("s.compareTo(p) = " + s.compareTo(p));
}
```

```
p.compareTo(s) = 0
```

a type-safe compare

generic type argument: "formal generic type"

using the formal generic type

generic instantiation: "actual concrete type"

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

typing: o is Person

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Student s) {
    return num - s.num;
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.compareTo(s) = " + p.compareTo(s));
  System.out.println("s.compareTo(p) = " + s.compareTo(p));
}
```

```
p.compareTo(s) = 0
s.compareTo(p) = 0
```

a type-safe compare

generic type argument: "formal generic type"

using the formal generic type

```
interface Comparable<T> {
  int compareTo(T o);
}
```

generic instantiation: "actual concrete type"

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

typing: o is Person

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Student s) {
    return num - s.num;
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.compareTo(s) = " + p.compareTo(s));
  System.out.println("s.compareTo(p) = " + s.compareTo(p));
}
```

```
p.compareTo(s) = 0
s.compareTo(p) = 0
```

why?

a type-safe compare

generic type argument: "formal generic type"

```
interface Comparable<T> {
  int compareTo(T o);
}
```

implements Comparable<Student> not allowed

"actual concrete type"

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

typing: o is Person

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Student s) {
    return num - s.num;
  }
}
```

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.compareTo(s) = " + p.compareTo(s));
  System.out.println("s.compareTo(p) = " + s.compareTo(p));
}
```

```
p.compareTo(s) = 0
s.compareTo(p) = 0
```

why?

a type-safe compare

generic type argument:
"formal generic type"

```
interface Comparable<T> {
  int compareTo(T o);
}
```

implements Comparable<Student> not allowed

"actual concrete type"

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
```

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Student s) {
```

Compile-time error:
"method does not override or implement a method from a supertype"

```
private void run() {
  Person  p = new Person("Alice");
  Student s = new Student(7, "Alice");
  System.out.println("p.compareTo(s) = " + p.compareTo(s));
  System.out.println("s.compareTo(p) = " + s.compareTo(p));
}
```

```
p.compareTo(s) = 0
s.compareTo(p) = 0
```

why?

a type-safe compare

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Person s) {
    return num - s.num;
  }
}
```

a type-safe compare

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Person s) {
    return num - s.num;
  }
}
```

a type-safe compare

```
interface Comparable<T> {
  int compareTo(T o);
}
```

```
public class Person implements Comparable<Person> {
  ..
  @Override
  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
public class Student extends Person {
  ..
  @Override
  int compareTo(Person s) {
    return num - s.num;
  }
}
```

Compile-time error:
"error: cannot find symbol
return this.num - s.num;
symbol: variable snum
location: variable s of type **Person**"

equals with different types

```
public class Person implements Comparable<Person> {
  ..
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }

  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
Person p = new Person("Alice");
p.equals("Alice");
a) type error
b) true
c) false
```

equals with different types

```
public class Person implements Comparable<Person> {
  ..
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }

  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
Person p = new Person("Alice");
p.equals("Alice");
a) type error
b) true
c) false
```

compareTo with different types

```
public class Person implements Comparable<Person> {
  ..
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }

  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
Person p = new Person("Alice");
p.compareTo("Alice");
a) type error
b) 0
c) ≠ 0
```

compareTo with different types

```
public class Person implements Comparable<Person> {
  ..
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }

  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
Person p = new Person("Alice");
p.compareTo("Alice");
a) type error
b) 0
c) ≠ 0
```

equals with null

```
public class Person implements Comparable<Person> {
  ..
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }

  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
Person p = new Person("Alice");
p.equals(null);
a) type error
b) exception
c) true
d) false
```

equals with null

```
public class Person implements Comparable<Person> {
  ..
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }

  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
Person p = new Person("Alice");
p.equals(null);
a) type error
b) exception
c) true
d) false
```

compareTo with null

```
public class Person implements Comparable<Person> {
  ..
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }

  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
Person p = new Person("Alice");
p.compareTo(null);
a) type error
b) exception
c) 0
d) ≠ 0
```

compareTo with null

```
public class Person implements Comparable<Person> {
  ..
  public boolean equals(Object obj) {
    if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    final Person q = (Person) obj;
    return name.equals(q.name);
  }

  public int compareTo(Person o) {
    return name.compareTo(o.name);
  }
}
```

```
Person p = new Person("Alice");
p.compareTo(null);
a) type error
b) exception
c) 0
d) ≠ 0
```

more uses of generic types: counting word frequency

store word and its count in array of pairs

1. update array for each word in input
2. sort array (lexicographically)
3. show array

more uses of generic types: counting word frequency

store word and its count in array of pairs

1. update array for each word in input
2. sort array (lexicographically)
3. show array

```
To be or not to be - that is the question!
```

more uses of generic types: counting word frequency

store word and its count in array of pairs

1. update array for each word in input
2. sort array (lexicographically)
3. show array

```
To be or not to be - that is the question!
```

```
Words:
be 2
is 1
not 1
or 1
question 1
that 1
the 1
to 2
```

make this map of words to count more reusable

many programs need pairs

- often other types than `String` and `int`

make this map of words to count more reusable

many programs need pairs

- often other types than `String` and `int`
- using `Object` instead of `String` and `int` spoils type safety

make this map of words to count more reusable

many programs need pairs

- often other types than `String` and `int`
- using `Object` instead of `String` and `int` spoils type safety

many programs need a `Map`

- not restricted to `Map` from `String` to `int`
  - `String` to `double` in expressions
  - `StudentNumber` to `Student`
  - `Zipcode` to `Address`
  - ..

make this map of words to count more reusable

many programs need pairs

- often other types than `String` and `int`
- using `Object` instead of `String` and `int` spoils type safety

many programs need a `Map`

- not restricted to `Map` from `String` to `int`
  - `String` to `double` in expressions
  - `StudentNumber` to `Student`
  - `Zipcode` to `Address`
  - ..

in Java we can make this more reusable by introducing type variables:
generic programming

make this map of words to count more reusable

many programs need pairs

- often other types than `String` and `int`
- using `Object` instead of `String` and `int` spoils type safety

many programs need a `Map`

- not restricted to `Map` from `String` to `int`
  - `String` to `double` in expressions
  - `StudentNumber` to `Student`
  - `Zipcode` to `Address`
  - ..

in Java we can make this more reusable by introducing type variables:
generic programming

- available since 2004 in JSE 5.0, SDK 1.5

reusable Pair

```
public class Pair <K, V> {
  private K key;
  private V val;
```

reusable Pair

```
public class Pair <K, V> {
  private K key;
  private V val;
```

K and V are generic type variables
typically a single uppercase letter

reusable Pair

```
public class Pair <K, V> {
  private K key;
  private V val;
```

K and V are generic type variables
typically a single uppercase letter

K and V are used like a type: attribute

reusable Pair

```
public class Pair <K, V> {
  private K key;
  private V val;

  public Pair(K key, V val) {
    this.key = key;
    this.val = val;
  }
```

K and V are generic type variables
typically a single uppercase letter

K and V are used like a type: attribute

reusable Pair

```
public class Pair <K, V> {
  private K key;
  private V val;

  public Pair(K key, V val) {
    this.key = key;
    this.val = val;
  }
```

K and V are generic type variables typically a single uppercase letter

K and V are used like a type: attribute

K and V are used like a type: argument of method

reusable Pair

```
public class Pair <K, V> {
  private K key;
  private V val;

  public Pair(K key, V val) {
    this.key = key;
    this.val = val;
  }

  public K getKey() { return key; }
  public V getVal() { return val; }
  public void setVal(V val) { this.val = val; }
}
```

K and V are generic type variables typically a single uppercase letter

K and V are used like a type: attribute

K and V are used like a type: argument of method

reusable Pair

```
public class Pair <K, V> {
  private K key;
  private V val;

  public Pair(K key, V val) {
    this.key = key;
    this.val = val;
  }

  public K getKey() { return key; }
  public V getVal() { return val; }
  public void setVal(V val) { this.val = val; }
}
```

K and V are generic type variables typically a single uppercase letter

K and V are used like a type: attribute

K and V are used like a type: argument of method

K and V are used like a type: result of method

allowed instances of type variable: any reference type

this **excludes** the **primitive types**

```
Pair<int, Student> p3 = new Pair<>(42, alice);
```

allowed instances of type variable: any reference type

this **excludes** the **primitive types**

this is **NOT** allowed !

```
Pair<int, Student> p3 = new Pair<>(42, alice);
```

allowed instances of type variable: any reference type

this **excludes** the **primitive types**

this is **NOT** allowed !

```
Pair<int, Student> p3 = new Pair<>(42, alice);
```

solution: use **wrapper types**

- these are predefined in Java:

  `int`, `double`, `char`, `boolean` wrapped in
  `Integer`, `Double`, `Character`, `Boolean`

allowed instances of type variable: any reference type

this **excludes** the **primitive types**

this is **NOT** allowed !

```
Pair<int, Student> p3 = new Pair<>(42, alice);
```

solution: use **wrapper types**

- these are predefined in Java:

  `int`, `double`, `char`, `boolean` wrapped in
  `Integer`, `Double`, `Character`, `Boolean`

`Integer` and `Double` are subtypes of `Number`

allowed instances of type variable: any reference type

this **excludes** the **primitive types**

this is **NOT** allowed !

```
Pair<int, Student> p3 = new Pair<>(42, alice);
```

solution: use **wrapper types**

- these are predefined in Java:

  `int`, `double`, `char`, `boolean` wrapped in
  `Integer`, `Double`, `Character`, `Boolean`

`Integer` and `Double` are subtypes of `Number`

use this as

```
Pair<Integer, Student> p3 = new Pair<>(42, alice);
```

allowed instances of type variable: any reference type

this **excludes** the **primitive types**

this is **NOT** allowed !

```
Pair<int, Student> p3 = new Pair<>(42, alice);
```

solution: use **wrapper types**

- these are predefined in Java:

  `int`, `double`, `char`, `boolean` wrapped in
  `Integer`, `Double`, `Character`, `Boolean`

`Integer` and `Double` are subtypes of `Number`

use this as

```
Pair<Integer, Student> p3 = new Pair<>(42, alice);
```

autoboxing / auto-unboxing: automatic conversion between primitive & wrapper

```
Integer box = 7;
int plain   = box;
```

allowed instances of type variable: any reference type

this **excludes** the **primitive types**

this is **NOT** allowed !

```
Pair<int, Student> p3 = new Pair<>(42, alice);
```

solution: use **wrapper types**

- these are predefined in Java:

  `int`, `double`, `char`, `boolean` wrapped in
  `Integer`, `Double`, `Character`, `Boolean`

`Integer` and `Double` are subtypes of `Number`

use this as

```
Pair<Integer, Student> p3 = new Pair<>(42, alice);
```

autoboxing / auto-unboxing: automatic c…er

```
Integer box = 7;
int plain   = box;
```

instead of

```
Integer box = new Integer(7);
int plain   = box.intValue();
```

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }
```

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }
```

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }

  public void add(String key) {
    for (Pair <String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        pair.setVal( pair.getVal() + 1 );
        return;
      }
    }
    map[count++] = new Pair<>(key, 1);
  }
```

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }

  public void add(String key) {
    for (Pair <String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        pair.setVal( pair.getVal() + 1 );
        return;
      }
    }
    map[count++] = new Pair<>(key, 1);
  }
```

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }

  public void add(String key) {
    for (Pair <String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        pair.setVal( pair.getVal() + 1 );
        return;
      }
    }
    map[count++] = new Pair<>(key, 1);
  }
```

actual types

diamond operator: deduced type
compiler must be able to deduce at compile-time

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }

  public void add(String key) {
    for (Pair <String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        pair.setVal( pair.getVal() + 1 );
        return;
      }
    }
    map[count++] = new Pair<>(key, 1);
  }
```

actual types

no diamond operator here ! ?

diamond operator: deduced type
compiler must be able to deduce at compile-time

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }

  public void add(String key) {
    for (Pair <String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        pair.setVal( pair.getVal() + 1 );
        return;
      }
    }
    map[count++] = new Pair<>(key, 1);
  }
```

note: creating an **Array of** Pairs

actual types

no diamond operator here ! ?

diamond operator: deduced type
compiler must be able to deduce at compile-time!

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }

  public void add(String key) {
    for (Pair <String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        pair.setVal( pair.getVal() + 1 );
        return;
      }
    }
    map[count++] = new Pair<>(key, 1);
  }
```

note: creating an **Array of** Pairs

actual types

no diamond operator here ! ?

diamond operator: deduced type
compiler must be able to deduce at compile-time!

note: creating a **Pair**

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }

  public void add(String key) {
    for (Pair <String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        pair.setVal( pair.getVal() + 1 );
        return;
      }
    }
    map[count++] = new Pair<>(key, 1);
  }
```

note: creating an **Array of** Pairs

actual types

no diamond operator here ! ?

diamond operator: deduced type
compiler must be able to deduce at compile-time!

note: creating a **Pair**

ignoring size problems

mapping words to count – 1 / 2

```
public class Map {
  private Pair <String, Integer> [] map;
  protected int count = 0;

  public Map(int size) {
    map = new Pair[size];
  }

  public void add(String key) {
    for (Pair <String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        pair.setVal(pair.getVal() + 1 );
        return;
      }
    }
    map[count++] = new Pair<>(key, 1);
  }
```

note: creating an **Array of** Pairs

actual types

no diamond operator here ! ?

postfix ++ returns the value *before* increment

diamond operator: deduced type
compiler must be able to deduce at compile-time!

note: creating a **Pair**

ignoring size problems

mapping words to count – 2 / 2

```
public int get(String key) {
    for (Pair<String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return 0;
  }
```

```
}
```

mapping words to count – 2 / 2

```
 public int get(String key) {
    for (Pair<String, Integer> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return 0;
  }

  public String [] keys () {
    String [] keys = new String[count];
    for (int i = 0; i < count; i += 1) {
      keys[i] = map[i].getKey();
    }
    Arrays.sort(keys);
    return keys;
  }
}
```

counting words

```
private void run() {
  Scanner scan = new Scanner(line).useDelimiter("\\W+");
  Map map = new Map(100);
  while (scan.hasNext()) {
    map.add( scan.next().toLowerCase() );
  }
  for (String key: map.keys()) {
    System.out.println(key + ": " + map.get(key));
  }
}
```

counting

one or more “non-word characters”

```
private void run() {
  Scanner scan = new Scanner(line).useDelimiter("\\W+");
  Map map = new Map(100);
  while (scan.hasNext()) {
    map.add( scan.next().toLowerCase() );
  }
  for (String key: map.keys()) {
    System.out.println(key + ": " + map.get(key));
  }
}
```

counting

one or more “non-word characters”

```
private void run() {
  Scanner scan = new Scanner(line).useDelimiter("\\W+");
  Map map = new Map(100);
  while (scan.hasNext()) {
    map.add( scan.next().toLowerCase() );
  }
  for (String key: map.keys()) {
    System.out.println(key + ": " + map.get(key));
  }
}
```

```
be: 2
is: 1
not: 1
or: 1
question: 1
that: 1
the: 1
to: 2
```

generics for a single method

often the generic variables belong to a class;

they can also belong to a single method

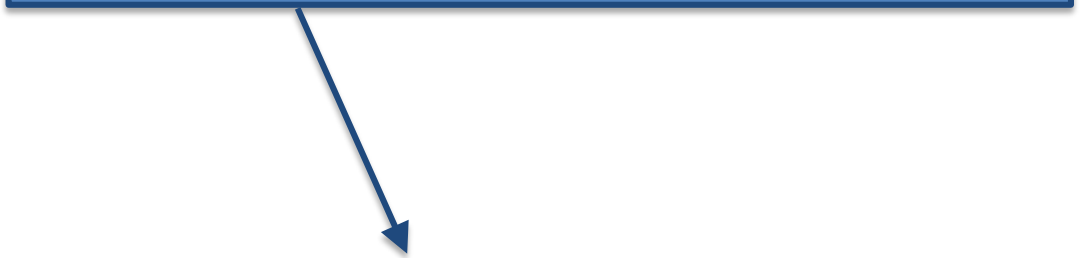
```
public <K,V> Pair<V,K> swap (Pair<K,V> p) {
  return new Pair<>(p.getVal(), p.getKey());
}
```

generics for a single method

often the generic variables belong to a class;

they can also belong to a single method

generic type arguments for method

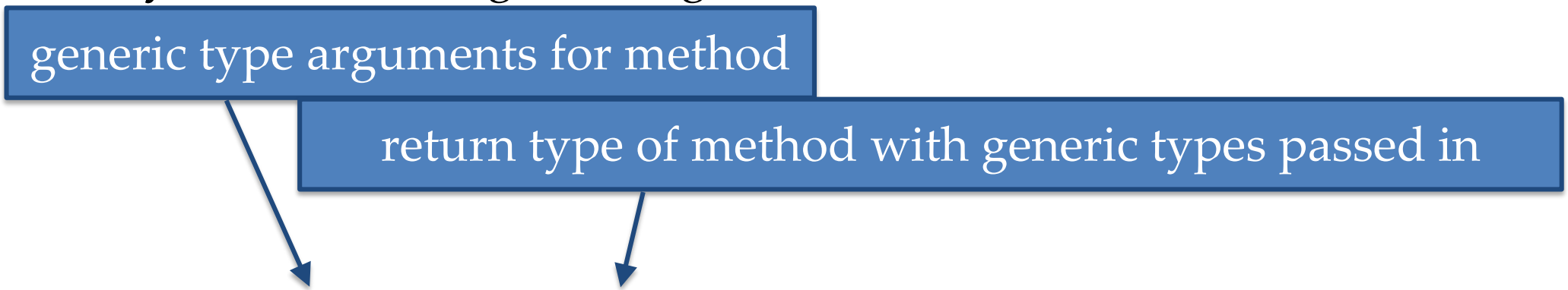


```
public <K,V> Pair<V,K> swap (Pair<K,V> p) {
  return new Pair<>(p.getVal(), p.getKey());
}
```

generics for a single method

often the generic variables belong to a class;

they can also belong to a single method

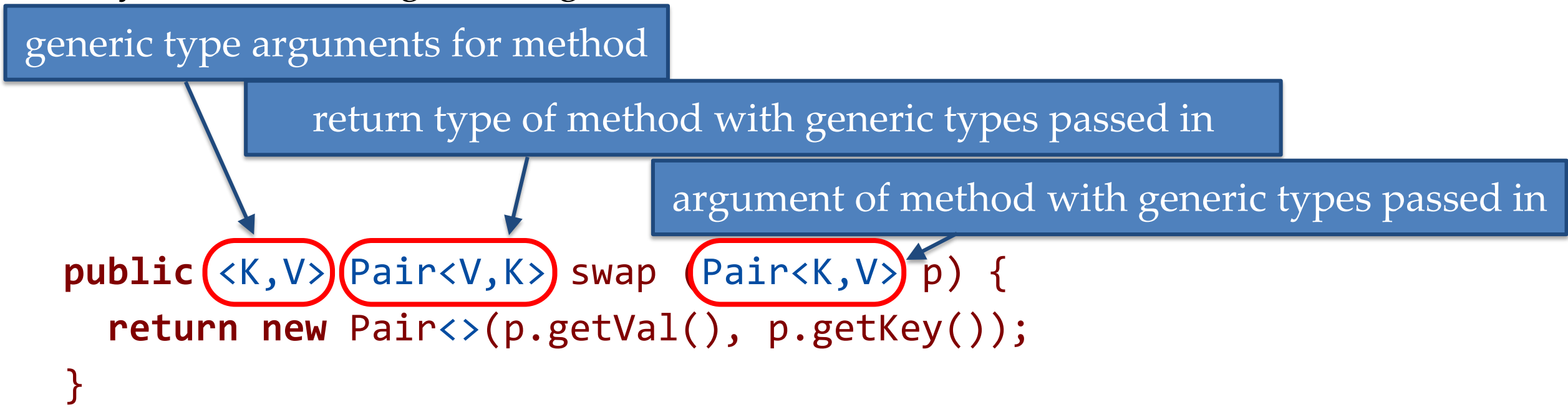



```
public <K,V> Pair<V,K> swap (Pair<K,V> p) {
  return new Pair<>(p.getVal(), p.getKey());
}
```

generics for a single method

often the generic variables belong to a class;

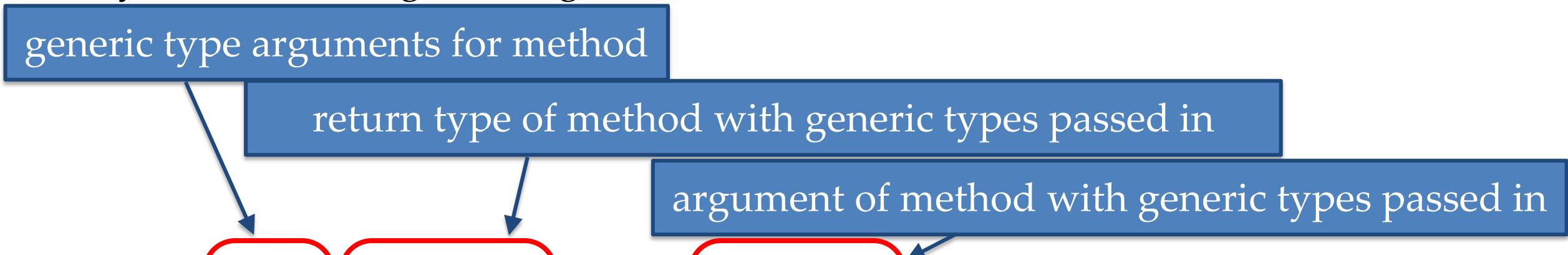
they can also belong to a single method



generics for a single method

often the generic variables belong to a class;

they can also belong to a single method




```
public <K,V> Pair<V,K> swap ( Pair<K,V> p) {
  return new Pair<>(p.getVal(), p.getKey());
}
```

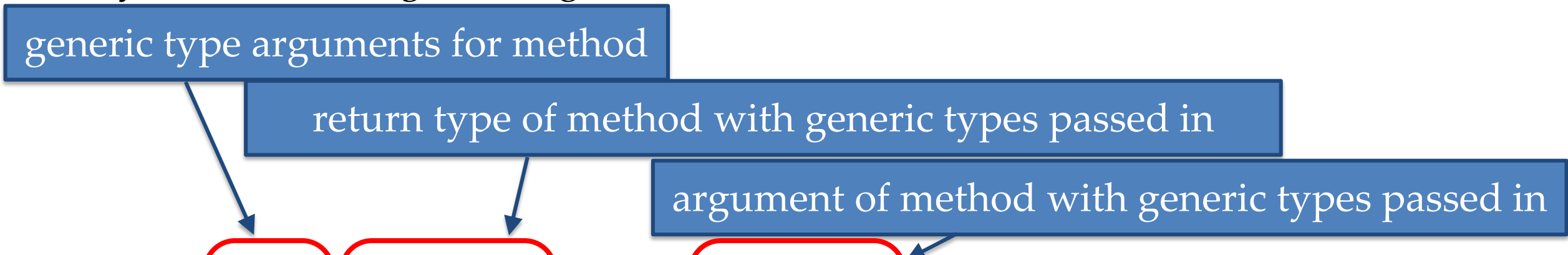
use this like any other method:

```
System.out.println(this.swap( pair ));
```

generics for a single method

often the generic variables belong to a class;

they can also belong to a single method




```
public <K,V> Pair<V,K> swap ( Pair<K,V> p) {
  return new Pair<>(p.getVal(), p.getKey());
}
```

use this like any other method:

```
System.out.println(this.swap( pair ));
```

or pass the type explicitly:

```
System.out.println(this.<String, Integer>swap( pair ));
```

type erasure

The one big limitation of Java generics:

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String>
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
                           T getX()
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
void setX(T x)
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
                T getX() → Object getX()
          void setX(T x) → void setX(Object x)
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
void setX(T x) → void setX(Object x)
ArrayList<String> list = new ArrayList<>();
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
void setX(T x) → void setX(Object x)
ArrayList<String> list = new ArrayList<>(); → ArrayList list = new ArrayList();
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
void setX(T x) → void setX(Object x)
ArrayList<String> list = new ArrayList<>(); → ArrayList list = new ArrayList();
String s = list.get(0);
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
void setX(T x) → void setX(Object x)
ArrayList<String> list = new ArrayList<>(); → ArrayList list = new ArrayList();
String s = list.get(0); → String s = (String)(list.get(0));
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
void setX(T x) → void setX(Object x)
ArrayList<String> list = new ArrayList<>(); → ArrayList list = new ArrayList();
String s = list.get(0); → String s = (String)(list.get(0));
<E extends Comparable> int e(E o1, E o2)...
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
void setX(T x) → void setX(Object x)
ArrayList<String> list = new ArrayList<>(); → ArrayList list = new ArrayList();
String s = list.get(0); → String s = (String)(list.get(0));
<E extends Comparable> int e(E o1, E o2)... → int e(Comparable o1,
                                                    Comparable o2) ...
```

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
void setX(T x) → void setX(Object x)
ArrayList<String> list = new ArrayList<>(); → ArrayList list = new ArrayList();
String s = list.get(0); → String s = (String)(list.get(0));
<E extends Comparable> int e(E o1, E o2)... → int e(Comparable o1,
                                                    Comparable o2) ...
```

conclusion: the **JVM** has **no knowledge** of the generic type information

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
                T getX() → Object getX()
           void setX(T x) → void setX(Object x)
ArrayList<String> list = new ArrayList<>(); → ArrayList list = new ArrayList();
          String s = list.get(0); → String s = (String)(list.get(0));
<E extends Comparable> int e(E o1, E o2)... → int e(Comparable o1,
                                                    Comparable o2) ...
```

conclusion: the **JVM** has **no knowledge** of the generic type information

at runtime there is no difference between
`GenPair<Integer, String>` and `GenPair<String, Integer>`:

type erasure

The one big limitation of Java generics:

generic types are checked at **compile-time**,

generic type information is **then removed**

```
GenPair<Integer, String> → GenPair
T getX() → Object getX()
void setX(T x) → void setX(Object x)
ArrayList<String> list = new ArrayList<>(); → ArrayList list = new ArrayList();
String s = list.get(0); → String s = (String)(list.get(0));
<E extends Comparable> int e(E o1, E o2)... → int e(Comparable o1,
                                                    Comparable o2) ...
```

conclusion: the **JVM** has **no knowledge** of the generic type information

at runtime there is no difference between `GenPair<Integer, String>` and `GenPair<String, Integer>`:

- both have type `GenPair`!

limitation of generic arguments

type parameter `<T>` cannot be used as a constructor (to create a new objects)

```
T object  = new T();
T[] array = new T[10];
```

limitation of generic arguments

type parameter `<T>` cannot be used as a constructor (to create a new objects)

```
T object  = new T();
T[] array = new T[10];
```

this is **NOT** allowed !

limitation of generic arguments

type parameter `<T>` cannot be used as a constructor (to create a new objects)

```
T object  = new T();
T[] array = new T[10];
```

this is **NOT** allowed !

no generic type parameters of a Class in static methods

limitation of generic arguments

type parameter `<T>` cannot be used as a constructor (to create a new objects)

```
T object  = new T();
T[] array = new T[10];
```

this is **NOT** allowed !

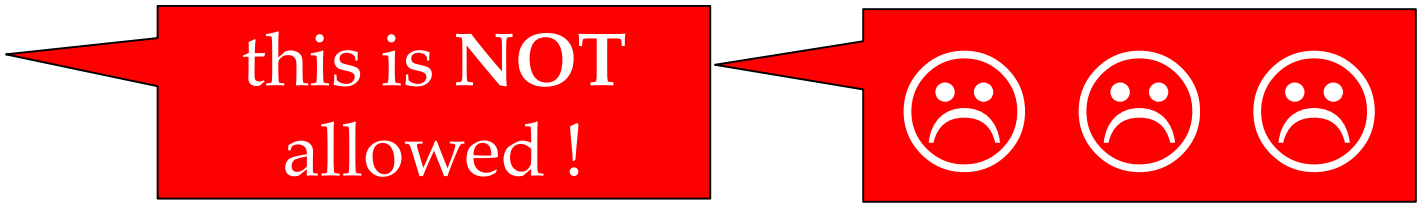
no generic type parameters of a Class in static methods

- (caveat: generic static methods that take generic type arguments are okay, e.g.
  `public static <K> void add (K o) { … } `)

limitation of generic arguments

type parameter `<T>` cannot be used as a constructor (to create a new objects)

```
T object  = new T();
T[] array = new T[10];
```



no generic type parameters of a Class in static methods

- (caveat: generic static methods that take generic type arguments are okay, e.g. `public static <K> void add (K o) { … } `)

no generic exceptions

a generic map 1 / 2

```
public class MapGen <K extends Comparable<K>, V> {
  private Pair <K, V> [] map;
  private int count = 0;

  public MapGen(int size) {
    map = new Pair[size];    ☹
  }
```

a generic map 1 / 2

```
public class MapGen <K extends Comparable<K>, V> {
  private Pair <K, V> [] map;
  private int count = 0;

  public MapGen(int size) {
    map = new Pair[size];     ☹
  }
```

bounded generic type:
to ensure that we can sort the pairs

a generic map 1 / 2

```
public class MapGen <K extends Comparable<K>, V> {
  private Pair <K, V> [] map;
  private int count = 0;

  public MapGen(int size) {
    map = new Pair[size];
  }
```

☹

bounded generic type:
to ensure that we can sort the pairs

`new Pair<>[size]` not allowed

a generic map 1 / 2

```
public class MapGen <K extends Comparable<K>, V> {
  private Pair <K, V> [] map;
  private int count = 0;

  public MapGen(int size) {
    map = new Pair[size];    ☹
  }

  public void add(K key, V val) {
    for (Pair<K, V> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        pair.setVal(val);
        return;
      }
    }
    map[count++] = new Pair<>(key, val);
  }
```

bounded generic type:
to ensure that we can sort the pairs

`new Pair<>[size]` not allowed

a generic map 2 / 2

```
  public V get(K key) {
    for (Pair<K, V> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return null;
  }









}
```

a generic map 2 / 2

```
  public V get(K key) {
    for (Pair<K, V> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return null;
  }

  public K [] keys () {
    K [] keys = (K[]) new Object[count];
    for (int i = 0; i < count; i += 1) {
      keys[i] = map[i].getKey();
    }
    Arrays.sort(keys);
    return keys;
  }
}
```

a generic map 2 / 2

```
  public V get(K key) {
    for (Pair<K, V> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return null;
  }

  public K [] keys () {
    K [] keys = (K[]) new Object[count];
    for (int i = 0; i < count; i += 1) {
      keys[i] = map[i].getKey();
    }
    Arrays.sort(keys);
    return keys;
  }
}
```

since `new K [count]` is not allowed

a generic map 2 / 2

```
  public V get(K key) {
    for (Pair<K, V> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return null;
  }

  public K [] keys () {
    K [] keys = (K[]) new Object[count];
    for (int i = 0; i < count; i += 1) {
      keys[i] = map[i].getKey();
    }
    Arrays.sort(keys);
    return keys;
  }
}
```

since `new K [count]` is not allowed

this gives an exception:
cannot cast `Object` to `Comparable`

a generic map 2 / 2

```
  public V get(K key) {
    for (Pair<K, V> pair : map) {
      if (pair != null && pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return null;
  }

  public K [] keys () {
    K [] keys = (K[]) new Object[count];
    for (int i = 0; i < count; i += 1) {
      keys[i] = map[i].getKey();
    }
    Arrays.sort(keys);
    return keys;
  }
}
```

since `new K [count]` is not allowed

this gives an exception:
cannot cast `Object` to `Comparable`

disappointing that the compiler does not spot this

fixing the generic map

return an `ArrayList<K>` instead of an `Array`

```
public ArrayList<K> keys () {
  ArrayList<K> keys = new ArrayList<>(count);
  for (int i = 0; i < count; i += 1) {
    keys.add( map[i].getKey() );
  }
  Collections.sort(keys);
  return keys;
}
```

fixing the generic map

return an `ArrayList<K>` instead of an `Array`

```
public ArrayList<K> keys () {
  ArrayList<K> keys = new ArrayList<>(count);
  for (int i = 0; i < count; i += 1) {
    keys.add( map[i].getKey() );
  }
  Collections.sort(keys);
  return keys;
}
```

we can always add an element to an `ArrayList`,
this is a much better solution too for `private Pair <K, V> [] map;`

fixing the generic map

return an `ArrayList<K>` instead of an `Array`

good general rule as well; ArrayLists are so much nicer to work with…

```
public ArrayList<K> keys () {
  ArrayList<K> keys = new ArrayList<>(count);
  for (int i = 0; i < count; i += 1) {
    keys.add( map[i].getKey() );
  }
  Collections.sort(keys);
  return keys;
}
```

we can always add an element to an `ArrayList`,
this is a much better solution too for `private Pair <K, V> [] map;`

fixing the generic map

return an `ArrayList<K>` instead of an `Array`

good general rule as well; ArrayLists are so much nicer to work with…

```
public ArrayList<K> keys () {
  ArrayList<K> keys = new ArrayList<>(count);
  for (int i = 0; i < count; i += 1) {
    keys.add( map[i].getKey() );
  }
  Collections.sort(keys);
  return keys;
}
```

we can always add an element to an `ArrayList`,
this is a much better solution too for `private Pair <K, V> [] map;`

counting words with the generic map

```
private void run2() {
  Scanner scan = new Scanner(line).useDelimiter("\\W+");
  MapGen<String, Integer> map = new MapGen<>(100);




}
```

counting words with the generic map

```
private void run2() {
  Scanner scan = new Scanner(line).useDelimiter("\\W+");
  MapGen<String, Integer> map = new MapGen<>(100);
  while (scan.hasNext()) {
    String word = scan.next().toLowerCase();
    try {
    Integer prevCount = map.get(word);
    if ( prevCount == null ) {
      map.add(word, 1);
    } else {
      map.add(word, prevCount + 1);
    }
  }



}
```

counting words with the generic map

```
private void run2() {
  Scanner scan = new Scanner(line).useDelimiter("\\W+");
  MapGen<String, Integer> map = new MapGen<>(100);
  while (scan.hasNext()) {
    String word = scan.next().toLowerCase();
    try {
    Integer prevCount = map.get(word);
    if ( prevCount == null ) {
      map.add(word, 1);
    } else {
      map.add(word, prevCount + 1);
    }
  }
  for (String key: map.keys()) {
    System.out.println(key + ": " + map.get(key));
  }
}
```

counting words with the generic map

```
private void run2() {
  Scanner scan = new Scanner(line).useDelimiter("\\W+");
  MapGen<String, Integer> map = new MapGen<>(100);
  while (scan.hasNext()) {
    String word = scan.next().toLowerCase();
    try {
    Integer prevCount = map.get(word);
    if ( prevCount == null ) {
      map.add(word, 1);
    } else {
      map.add(word, prevCount + 1);
    }
  }
  for (String key: map.keys()) {
    System.out.println(key + ": " + map.get(key));
  }
}
```

```
be: 2
is: 1
not: 1
or: 1
question: 1
that: 1
the: 1
to: 2
```

limitations on subtypes: make subclass

```
class Person {
  protected String name;
  public Person(String name) { this.name = name; }
  @Override
  public String toString() {
    return "Person " + name;
  }
}
```

limitations on subtypes: make subclass

```
class Person {
  protected String name;
  public Person(String name) { this.name = name; }
  @Override
  public String toString() {
    return "Person " + name;
  }
}
class Student extends Person {
  protected int num;
  public Student(int num, String name) {
    super(name);
    this.num = num;
  }
  @Override
  public String toString() {
    return "Student " + name + " " + num;
  }
}
```

limitations on subtypes: this is fine

```
class Box <T> {
  private T e;
  public Box(T e) { this.e = e; }
  @Override
  public String toString() {
    return "Box{" + e + '}';
  }
}
```

limitations on subtypes: this is fine

```
class Box <T> {
  private T e;
  public Box(T e) { this.e = e; }
  @Override
  public String toString() {
    return "Box{" + e + '}';
  }
}
```

in some class:

```
private void run1 () {
  Box<Person> b1 = new Box<>(new Person("Alice"));
  Box<Person> b2 = new Box<>(new Student(1, "Bob"));
  System.out.println(b1 + " " + b2);
}
```

limitations on subtypes: this is fine

```
class Box <T> {
  private T e;
  public Box(T e) { this.e = e; }
  @Override
  public String toString() {
    return "Box{" + e + '}';
  }
}
```

in some class:

```
private void run1 () {
  Box<Person> b1 = new Box<>(new Person("Alice"));
  Box<Person> b2 = new Box<>(new Student(1, "Bob"));
  System.out.println(b1 + " " + b2);
}
```

b2 contains a subtype of Person

limitations on subtypes: this is fine

```
class Box <T> {
  private T e;
  public Box(T e) { this.e = e; }
  @Override
  public String toString() {
    return "Box{" + e + '}';
  }
}
```

in some class:

```
private void run1 () {
  Box<Person> b1 = new Box<>(new Person("Alice"));
  Box<Person> b2 = new Box<>(new Student(1, "Bob"));
  System.out.println(b1 + " " + b2);
}
```

b2 contains a subtype of Person

```
Box{Person Alice} Box{Student Bob 1}
```

limitations on subtypes: still fine

```
private void print(Box<Person> b) {
  System.out.print(b);
}

private void run1 () {
  Box<Person> b1 = new Box<>(new Person("Alice"));
  Box<Person> b2 = new Box<>(new Student (1, "Bob"));
  print(b1);
  print(b2);
}
```

limitations on subtypes: still fine

```
private void print(Box<Person> b) {
  System.out.print(b);
}

private void run1 () {
  Box<Person> b1 = new Box<>(new Person("Alice"));
  Box<Person> b2 = new Box<>(new Student (1, "Bob"));
  print(b1);
  print(b2);
}
```

limitations on subtypes: still fine

```
private void print(Box<Person> b) {
  System.out.print(b);
}

private void run1 () {
  Box<Person> b1 = new Box<>(new Person("Alice"));
  Box<Person> b2 = new Box<>(new Student (1, "Bob"));
  print(b1);
  print(b2);
}
```

works fine ☺

limitations on subtypes: still fine

```
private void print(Box<Person> b) {
  System.out.print(b);
}

private void run1 () {
  Box<Person> b1 = new Box<>(new Person("Alice"));
  Box<Person> b2 = new Box<>(new Student (1, "Bob"));
  print(b1);
  print(b2);
}
```

works fine ☺

```
Box{Person Alice}
Box{Student Bob 1}
```

limitations on subtypes: this doesn't work

```
private void print(Box<Person> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

limitations on subtypes: this doesn't work

```
private void print(Box<Person> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

limitations on subtypes: this doesn't work

```
private void print(Box<Person> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

limitations on subtypes: this doesn't work

```
private void print(Box<Person> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

unchanged

error: incompatible types: Box<Student> cannot be converted to Box<Person>

limitations on subtypes: this doesn't work

```
private void print(Box<Person> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

unchanged

error: incompatible types: Box<Student> cannot be converted to Box<Person>

Solution: Wildcard Generic Types

wildcard generic types: bounded

```
private void print(Box<? extends Person> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

wildcard generic types: bounded

```
private void print(Box<? extends Person> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

wildcard generic types: bounded

```
private void print(Box<? extends Person> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

bounded wildcard generic

works fine ☺

wildcard generic types: bounded

```
private void print(Box<? extends Person> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

bounded wildcard generic

works fine ☺

```
Box{Person Alice}
Box{Student Bob 1}
```

wildcard generic types: lower bound

```
private void print(Box<? super Student> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

wildcard generic types: lower bound

```
private void print(Box<? super Student> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

lower bound wildcard generic (weird example, see book chapter 19.7 for details!)

wildcard generic types: lower bound

```
private void print(Box<? super Student> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

lower bound wildcard generic (weird example, see book chapter 19.7 for details!)

works fine ☺

wildcard generic types: lower bound

```
private void print(Box<? super Student> b) {
  System.out.print(b);
}

private void run2 () {
  Box<Person> b1  = new Box<>(new Person("Alice"));
  Box<Student> b2 = new Box<>(new Student(1,"Bob"));
  print(b1);
  print(b2);
}
```

lower bound wildcard generic (weird example, see book chapter 19.7 for details!)

works fine ☺

```
Box{Person Alice}
Box{Student Bob 1}
```

JOKE OF THE WEEK

a computer science student…

…was writing a note to his crush before lecture.
The student next to him grabbed the note.

student 1: You can't see that, it's private!

student 2: But we're in the same class!

providing similar actions for various container types

COLLECTIONS

the class Arrays

standard data types often need the same manipulations

- sorting, searching, equality, ..

the class Arrays

standard data types often need the same manipulations

- sorting, searching, equality, ..

`Arrays` provides useful operations on arbitrary arrays

- `fill`, `sort`, `binarySearch`, `equals, ..`
- we used the `Arrays.sort` in the assignments and above

the class Arrays

standard data types often need the same manipulations

- sorting, searching, equality, ..

`Arrays` provides useful operations on arbitrary arrays

- `fill`, `sort`, `binarySearch`, `equals, ..`
- we used the `Arrays.sort` in the assignments and above

`Arrays` does ***not*** box an ordinary array;

- it has no attributes
- static methods of `Arrays` take the array as argument
  e.g. `Arrays.sort( words );`

sometimes the size of a data type is unknown

`Strings` have a fixed length

`StringBuffers` can be changed

- it is always possible to add a character

sometimes the size of a data type is unknown

`Strings` have a fixed length

`StringBuffers` can be changed

- it is always possible to add a character

ordinary arrays have a fixed length

sometimes the size of a data type is unknown

`Strings` have a fixed length

`StringBuffers` can be changed

- it is always possible to add a character

ordinary arrays have a fixed length

`ArrayList` and **`LinkedList`** have a variable length

- it is always possible to add an element,
  at any place
- we can remove an element without the need to shift elements
- in contrast to **`Arrays`** these are real classes containing the element
- these classes implement the interface **`List<T>`**

a map of arbitrary size 1 / 2

```
public class MapList <K extends Comparable<K>, V> {
  protected List <Pair <K, V>> map;
```

a map of arbitrary size 1 / 2

```
public class MapList <K extends Comparable<K>, V> {
  protected List <Pair <K, V>> map;
  public MapList() {
    map = new ArrayList<>();
  }
```

a map of arbitrary size 1 / 2

```
public class MapList <K extends Comparable<K>, V> {
  protected List <Pair <K, V>> map;
  public MapList() {
    map = new ArrayList<>();
  }
  public void add(K key, V val) {
    for (Pair<K, V> pair : map) {
      if (pair.getKey().equals(key)) {
        pair.setVal(val);
        return;
      }
    }
    map.add(new Pair<>(key, val));
  }
```

a map of arbitrary size 1 / 2

```
public class MapList <K extends Comparable<K>, V> {
  protected List <Pair <K, V>> map;
  public MapList() {
    map = new ArrayList<>();
  }
  public void add(K key, V val) {
    for (Pair<K, V> pair : map) {
      if (pair.getKey().equals(key)) {
        pair.setVal(val);
        return;
      }
    }
    map.add(new Pair<>(key, val));
  }
```

no length limit

a map of arbitrary size 1 / 2

```
public class MapList <K extends Comparable<K>, V> {
  protected List <Pair <K, V>> map;
  public MapList() {
    map = new ArrayList<>();
  }
  public void add(K key, V val) {
    for (Pair<K, V> pair : map) {
      if (pair.getKey().equals(key)) {
        pair.setVal(val);
        return;
      }
    }
    map.add(new Pair<>(key, val));
  }
```

no length limit

no test for null

a map of arbitrary size 1 / 2

```
public class MapList <K extends Comparable<K>, V> {
  protected List <Pair <K, V>> map;
  public MapList() {
    map = new ArrayList<>();
  }
  public void add(K key, V val) {
    for (Pair<K, V> pair : map) {
      if (pair.getKey().equals(key)) {
        pair.setVal(val);
        return;
      }
    }
    map.add(new Pair<>(key, val));
  }
```

no length limit

no test for null

always fits

a map of arbitrary size 1 / 2

```
public class MapList <K extends Comparable<K>, V> {
  protected List <Pair <K, V>> map;
  public MapList() {
    map = new ArrayList<>();
  }
  public void add(K key, V val) {
    for (Pair<K, V> pair : map) {
      if (pair.getKey().equals(key)) {
        pair.setVal(val);
        return;
      }
    }
    map.add(new Pair<>(key, val));
  }
```

List is interface

no length limit

no test for null

always fits

a map of arbitrary size 1 / 2

```
public class MapList <K extends Comparable<K>, V> {
  protected List <Pair <K, V>> map;
  public MapList() {
    map = new ArrayList<>();
  }
  public void add(K key, V val) {
    for (Pair<K, V> pair : map) {
      if (pair.getKey().equals(key)) {
        pair.setVal(val);
        return;
      }
    }
    map.add(new Pair<>(key, val));
  }
```

List is interface

no length limit

ArrayList is class implementing **List**

no test for null

always fits

a map of arbitrary size 2 / 2

```
public V get(K key) {
  for (Pair<K, V> pair : map)
    if (pair.getKey().equals(key))
      return pair.getVal();
  return null;
}
```

a map of arbitrary size 2 / 2

```
  public V get(K key) {
    for (Pair<K, V> pair : map)
      if (pair.getKey().equals(key))
        return pair.getVal();
    return null;
  }
  public List<K> keys () {
    List<K> keys = new ArrayList<>();
    for (Pair<K, V> pair : map)
      keys.add(pair.getKey());
    Collections.sort(keys);
    return keys;
  }
}
```

a map of arbitrary size 2 / 2

```
  public V get(K key) {
    for (Pair<K, V> pair : map)
      if (pair.getKey().equals(key))
        return pair.getVal();
    return null;
  }
  public List<K> keys () {
    List<K> keys = new ArrayList<>();
    for (Pair<K, V> pair : map)
      keys.add(pair.getKey());
    Collections.sort(keys);
    return keys;
  }
}
```

Collections instead of **Arrays**

the interface `Collection`

we have several containers in Java

- `String`, `StringBuffer`, `ArrayList`, `LinkedList`, `Vector`, `Set`

many similar operations on these containers

- `isEmpty`, `contains`, `equals`, `size`

the interface `Collection`

we have several containers in Java

- `String`, `StringBuffer`, `ArrayList`, `LinkedList`, `Vector`, `Set`

many similar operations on these containers

- `isEmpty`, `contains`, `equals`, `size`

the interface **`Collection`** yields a uniform way to handle these kind of operations

the interface `Collection`

we have several containers in Java

- `String`, `StringBuffer`, `ArrayList`, `LinkedList`, `Vector`, `Set`

many similar operations on these containers

- `isEmpty`, `contains`, `equals`, `size`

the interface **`Collection`** yields a uniform way to handle these kind of operations

warning: there is also a **class** `Collections`

the interface `Collection`

we have several containers in Java

- `String`, `StringBuffer`, `ArrayList`, `LinkedList`, `Vector`, `Set`

many similar operations on these containers

- `isEmpty`, `contains`, `equals`, `size`

the interface **`Collection`** yields a uniform way to handle these kind of operations

warning: there is also a **class** `Collections`

- **`Collections`** is similar to **`Arrays`**: useful manipulations using static methods

the interface `Collection`

we have several containers in Java

- `String`, `StringBuffer`, `ArrayList`, `LinkedList`, `Vector`, `Set`

many similar operations on these containers

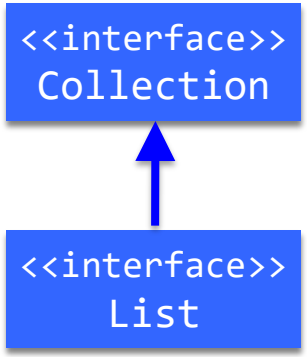
- `isEmpty`, `contains`, `equals`, `size`

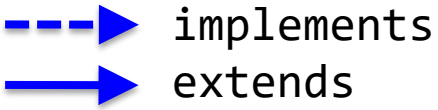
the interface **`Collection`** yields a uniform way to handle these kind of operations

warning: there is also a **class** `Collections`

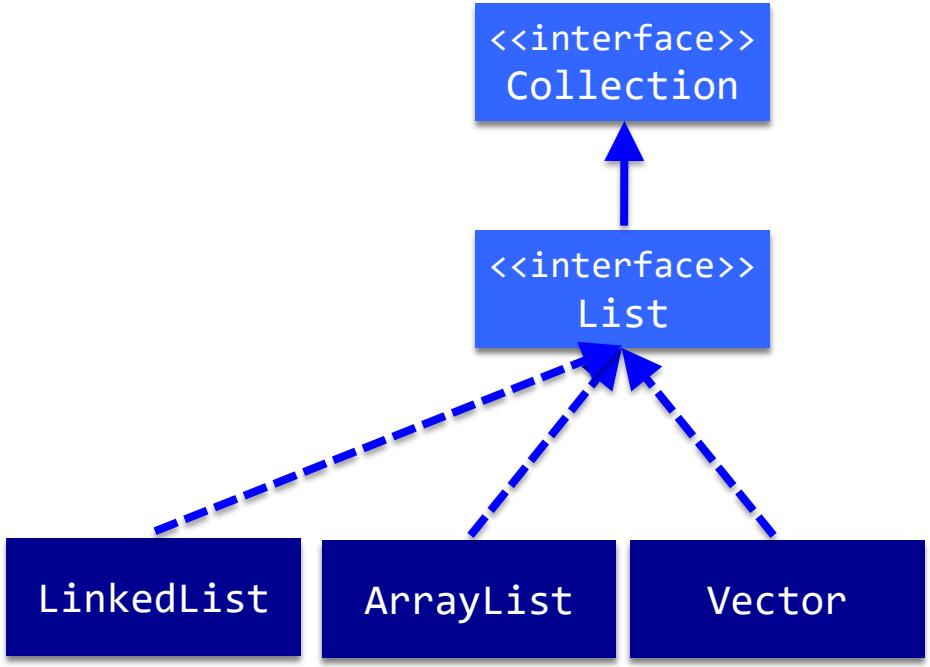
- **`Collections`** is similar to **`Arrays`**: useful manipulations using static methods
- don't confuse them

Collection Interface family





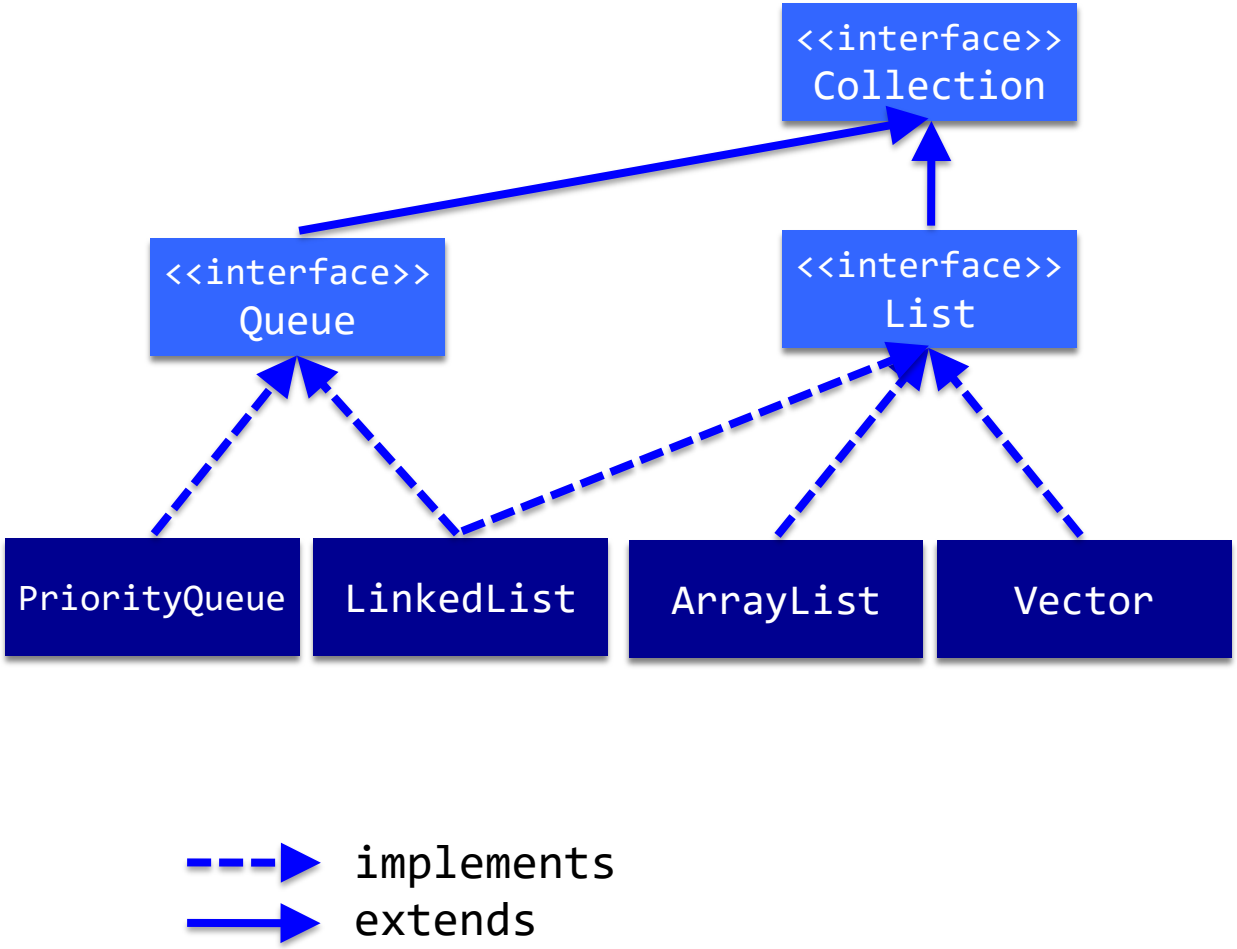
Collection Interface family



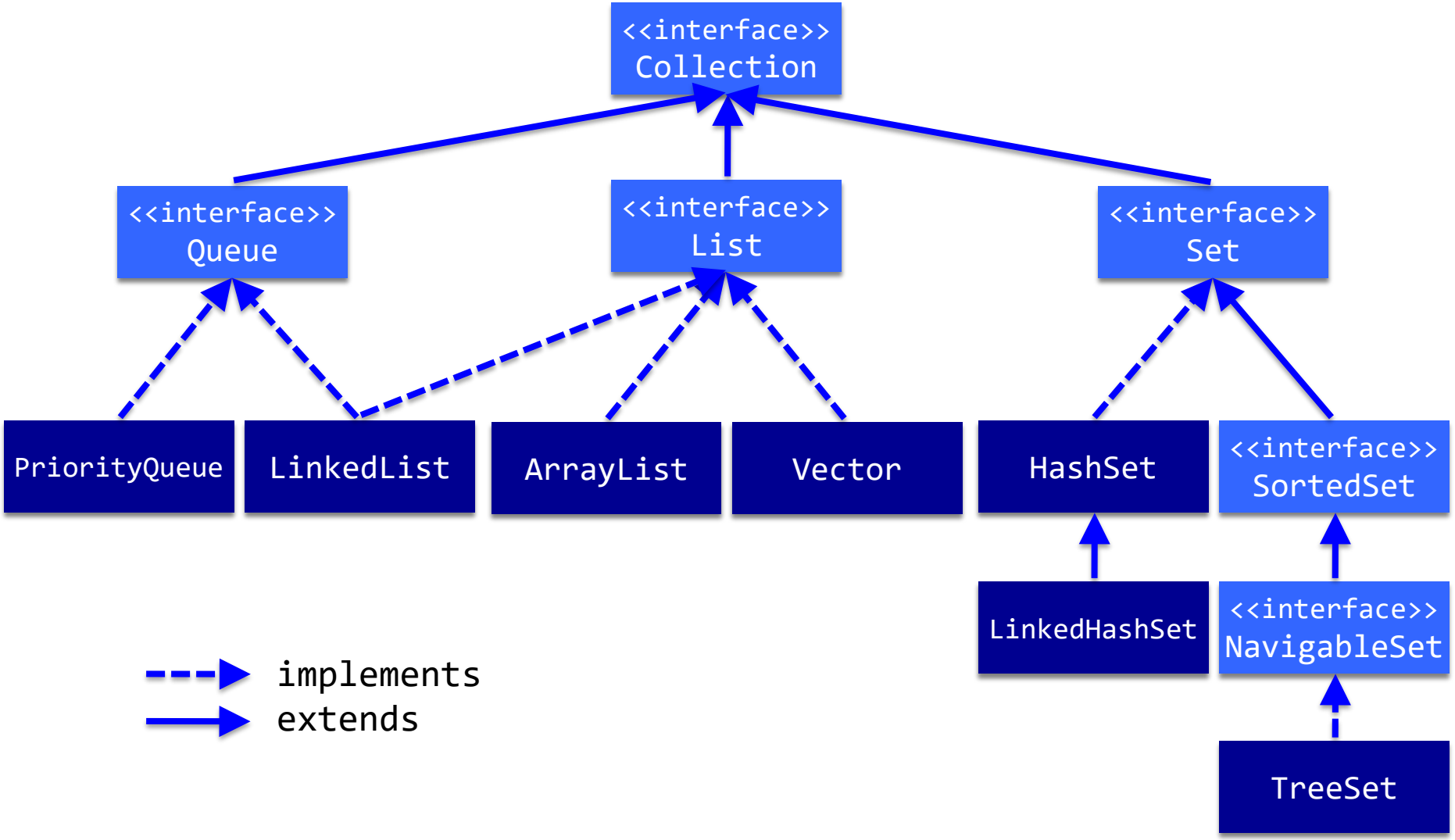
implements

extends

Collection Interface family

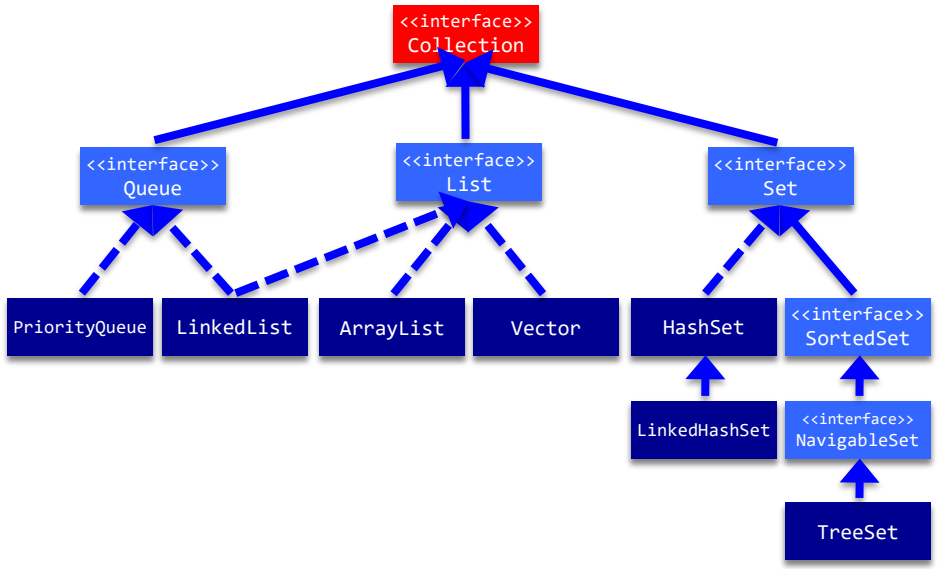


Collection Interface family



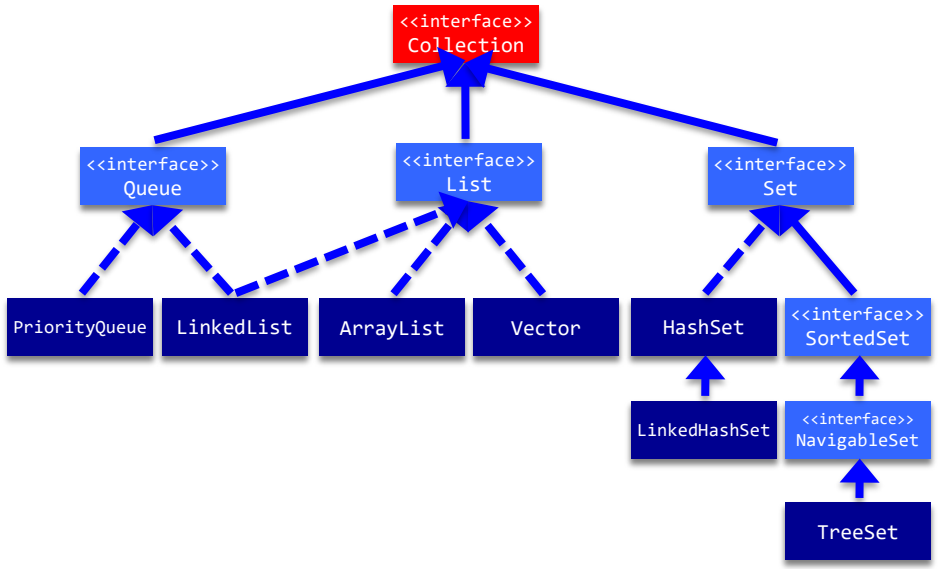
main methods in Interface Collection<E>

```
boolean add(E e)
```



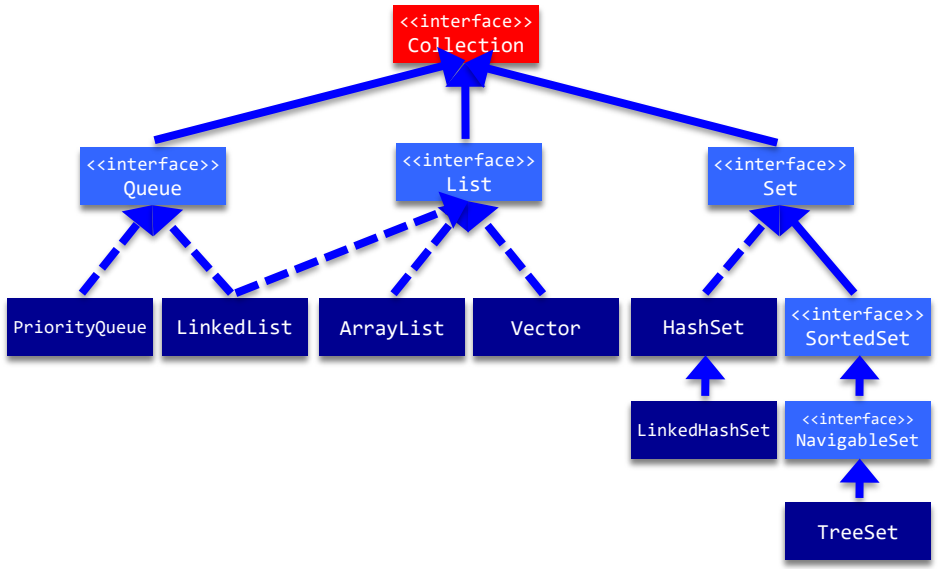
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
```



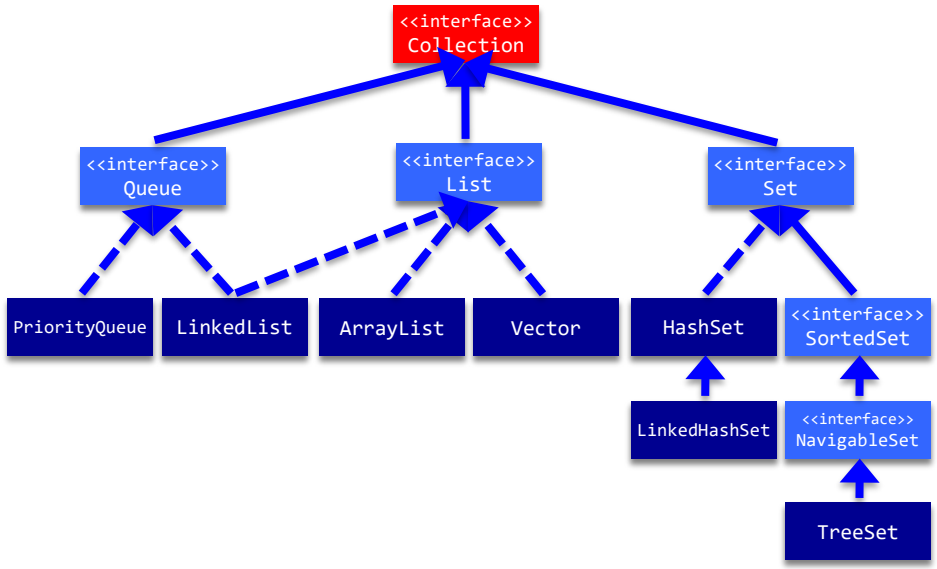
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
```



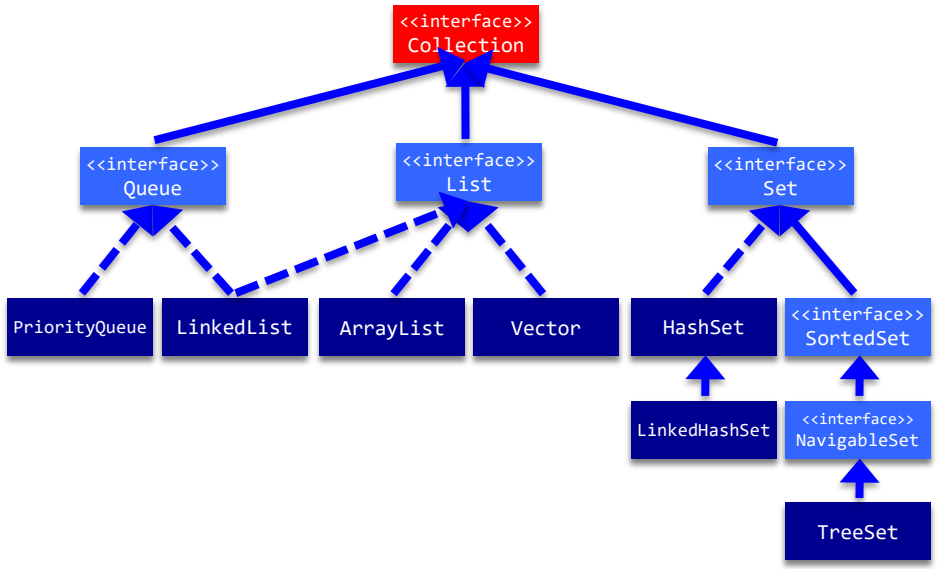
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
```



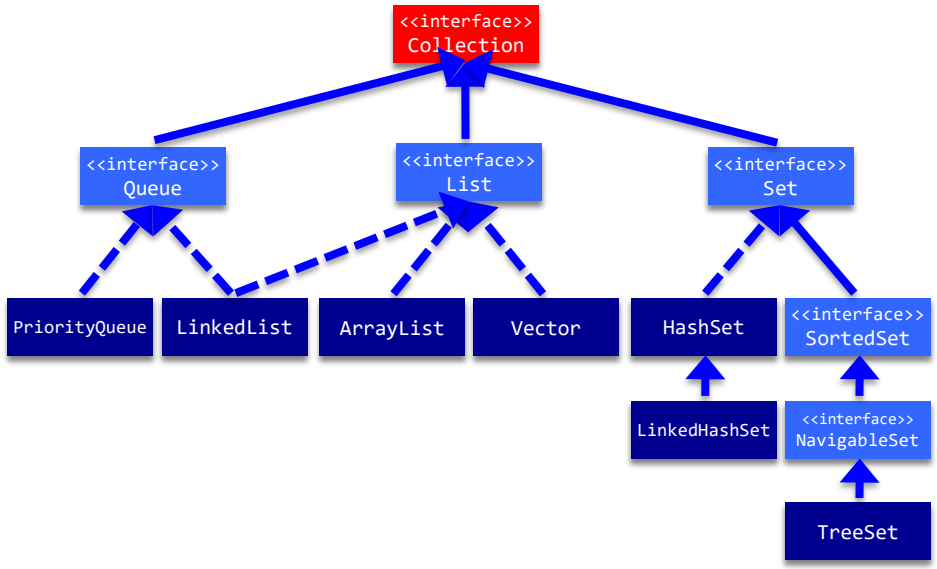
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
```



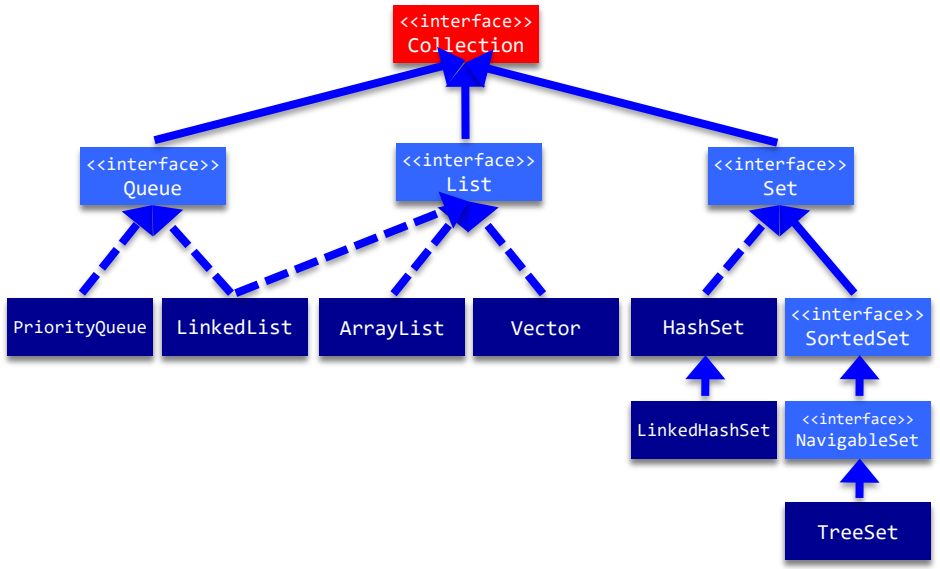
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
```



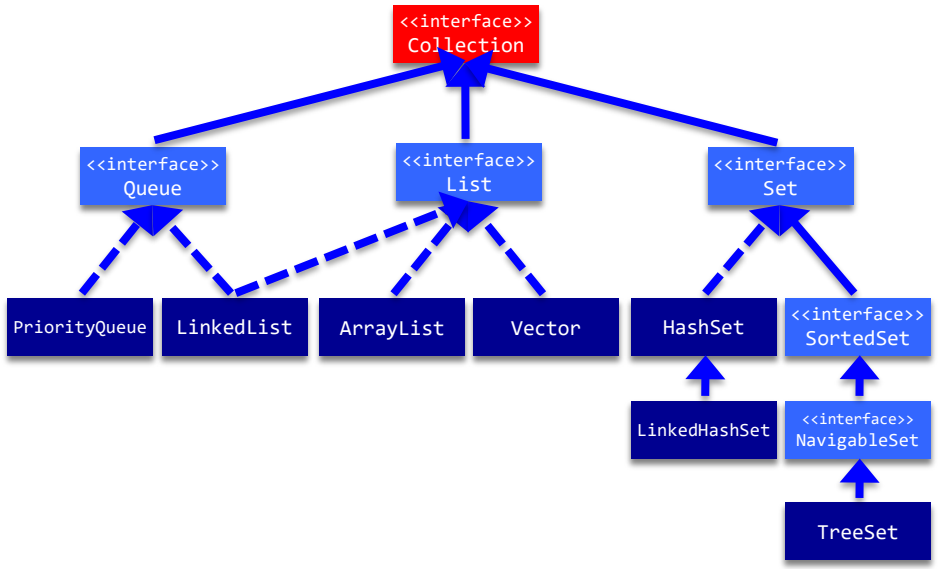
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
```



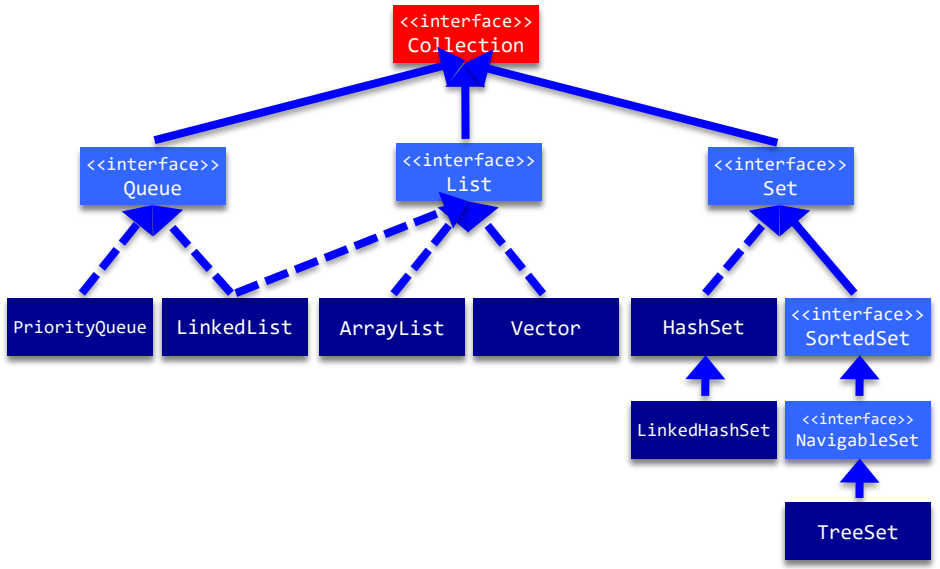
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
```



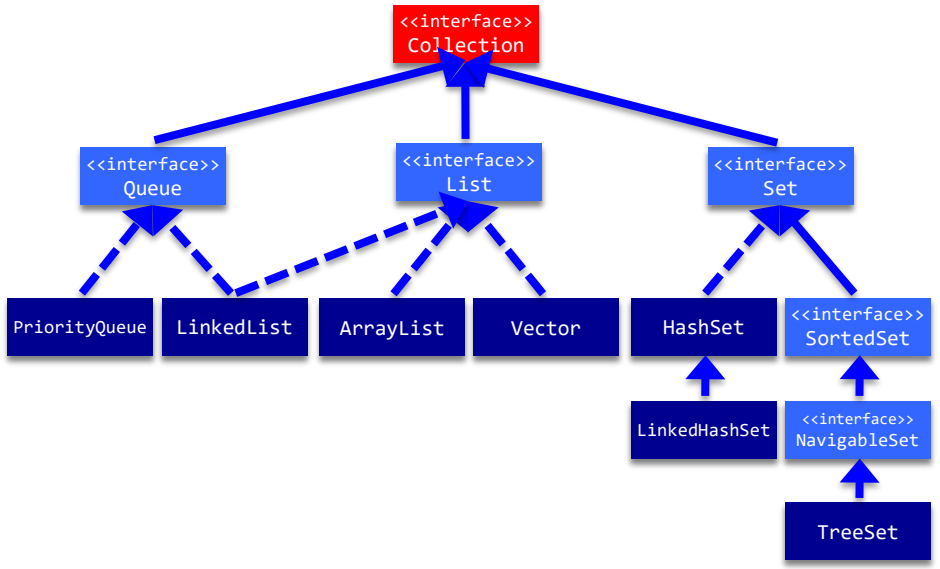
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
Iterator<E> iterator()
```



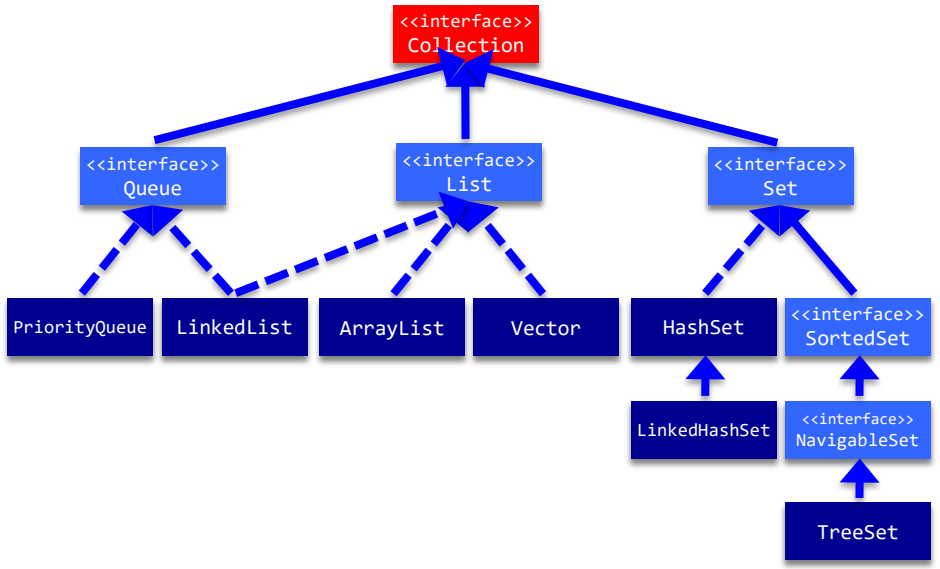
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
```



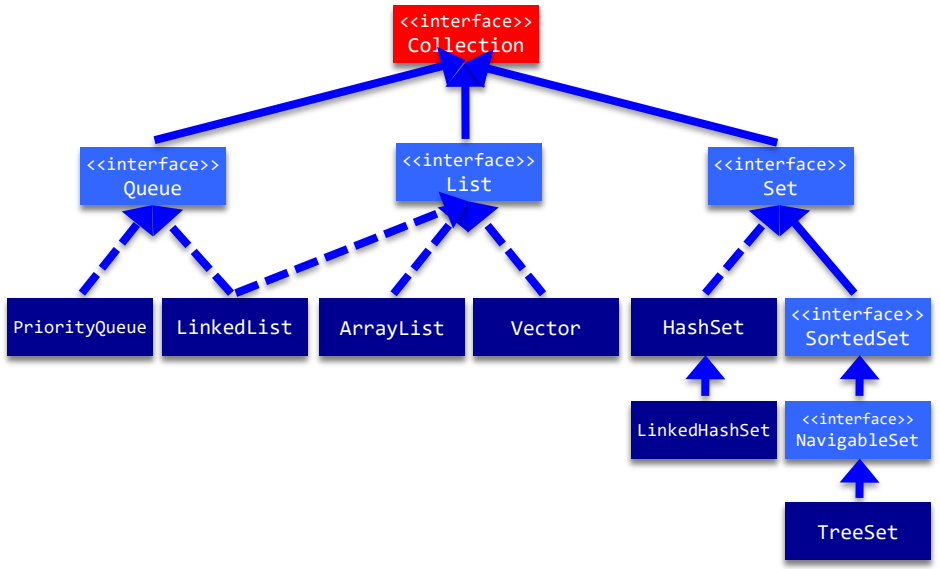
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
```



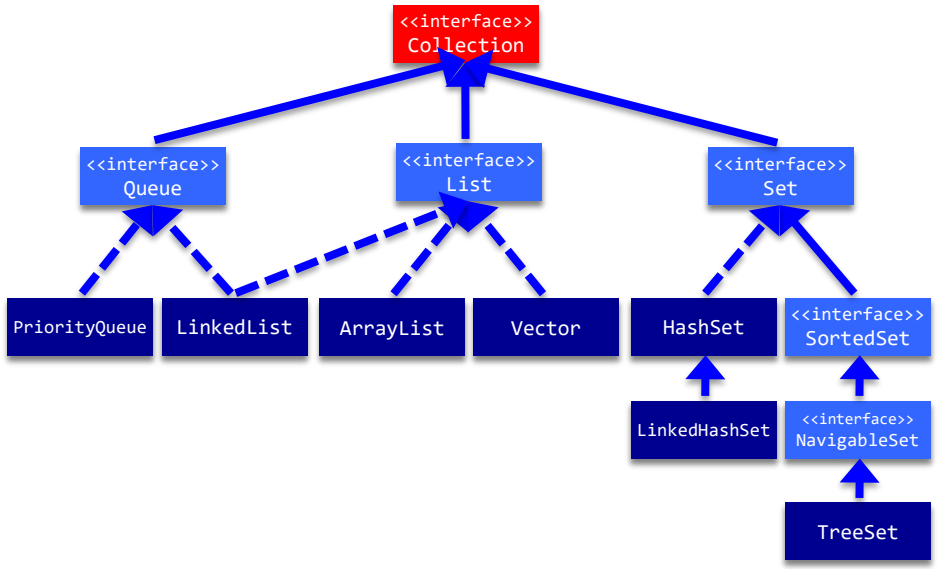
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
```



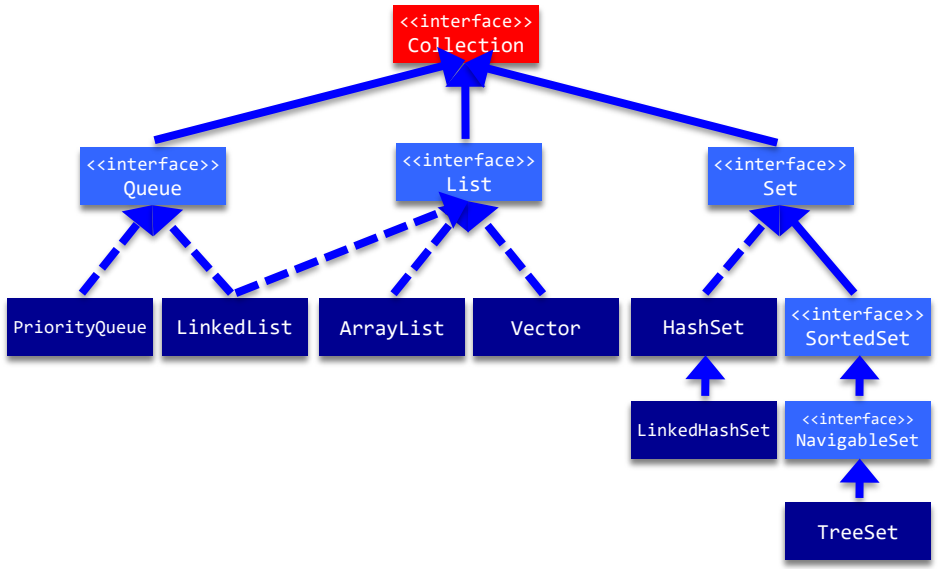
main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int     size()
```



main methods in Interface Collection<E>

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int     size()
<T> T[] toArray(T[] a)
```



iterators

`Iterator` offers a standard way to scan and handle all elements of a collection

- this is an interface; many implementations possible

the `Iterator` keeps track of the current element in a collection

there are methods to advance to the next element and to change the collection

iterators

Iterator offers a standard way to scan and handle all elements of a collection

- this is an interface; many implementations possible

the **Iterator** keeps track of the current element in a collection

there are methods to advance to the next element and to change the collection

```
Interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

iterators

Iterator offers a standard way to scan and handle all elements of a collection

- this is an interface; many implementations possible

the **Iterator** keeps track of the current element in a collection

there are methods to advance to the next element and to change the collection

```
Interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

is there a next object?

iterators

Iterator offers a standard way to scan and handle all elements of a collection

- this is an interface; many implementations possible

the **Iterator** keeps track of the current element in a collection

there are methods to advance to the next element and to change the collection

```
Interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

is there a next object?

yield next object; advance

iterators

Iterator offers a standard way to scan and handle all elements of a collection

- this is an interface; many implementations possible

the **Iterator** keeps track of the current element in a collection

there are methods to advance to the next element and to change the collection

```
Interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

is there a next object?

yield next object; advance

remove last object shown

iterators

`Iterator` offers a standard way to scan and handle all elements of a collection

- this is an interface; many implementations possible

the **`Iterator`** keeps track of the current element in a collection

there are methods to advance to the next element and to change the collection

```
Interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

is there a next object?

yield next object; advance

remove last object shown

optional operation, can throw a `NotImplementedException`

iterators

`Iterator` offers a standard way to scan and handle all elements of a collection

- this is an interface; many implementations possible

the **`Iterator`** keeps track of the current element in a collection

there are methods to advance to the next element and to change the collection

```
Interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

is there a next object?

yield next object; advance

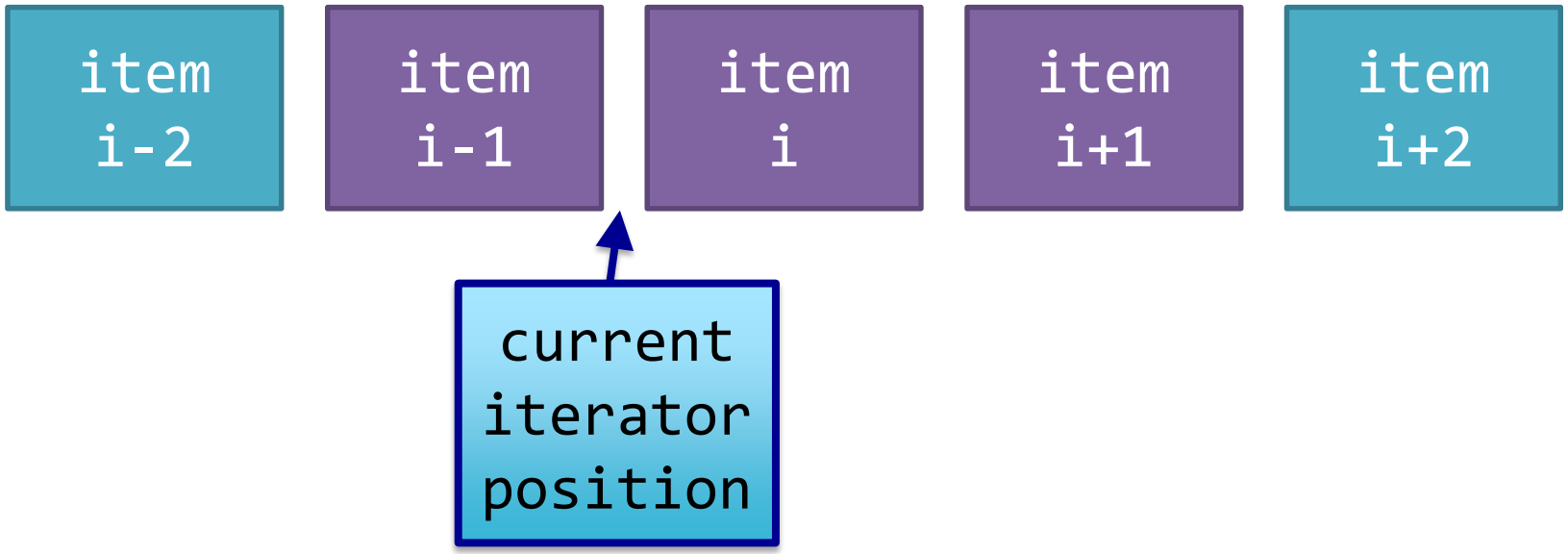
remove last object shown

optional operation, can throw a `NotImplementedException`

note: no `previous()`!

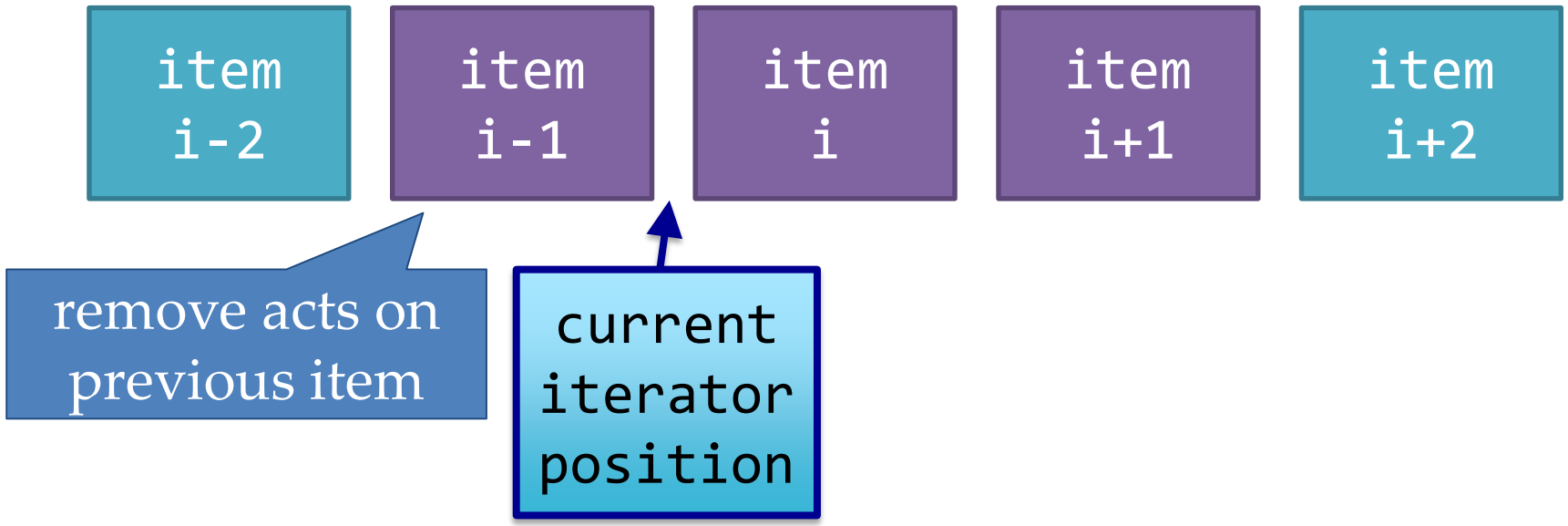
`Iterator` interface

an **`Iterator`** is conceptually *between* elements;
it does not refer to a particular object



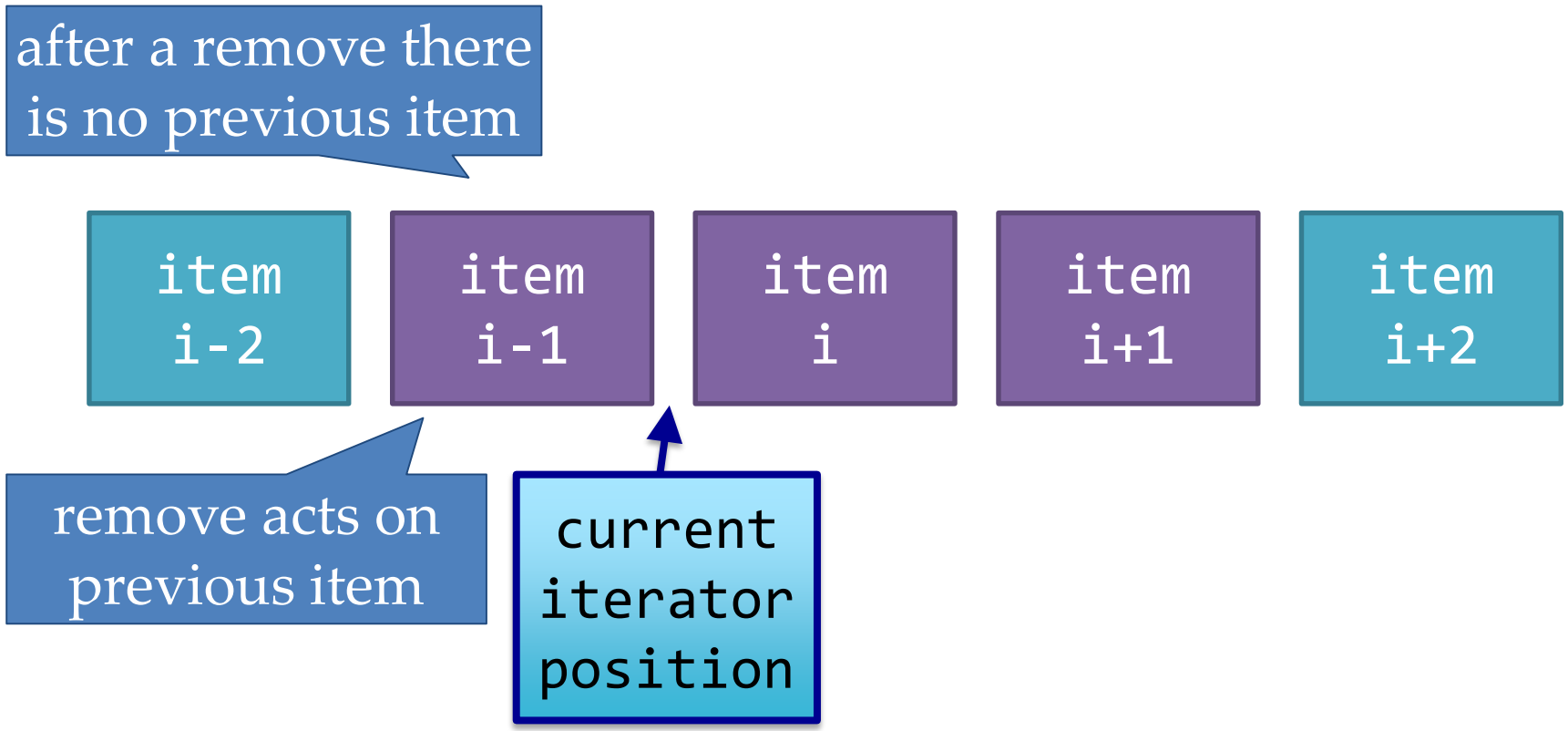
`Iterator` interface

an **`Iterator`** is conceptually *between* elements;
it does not refer to a particular object



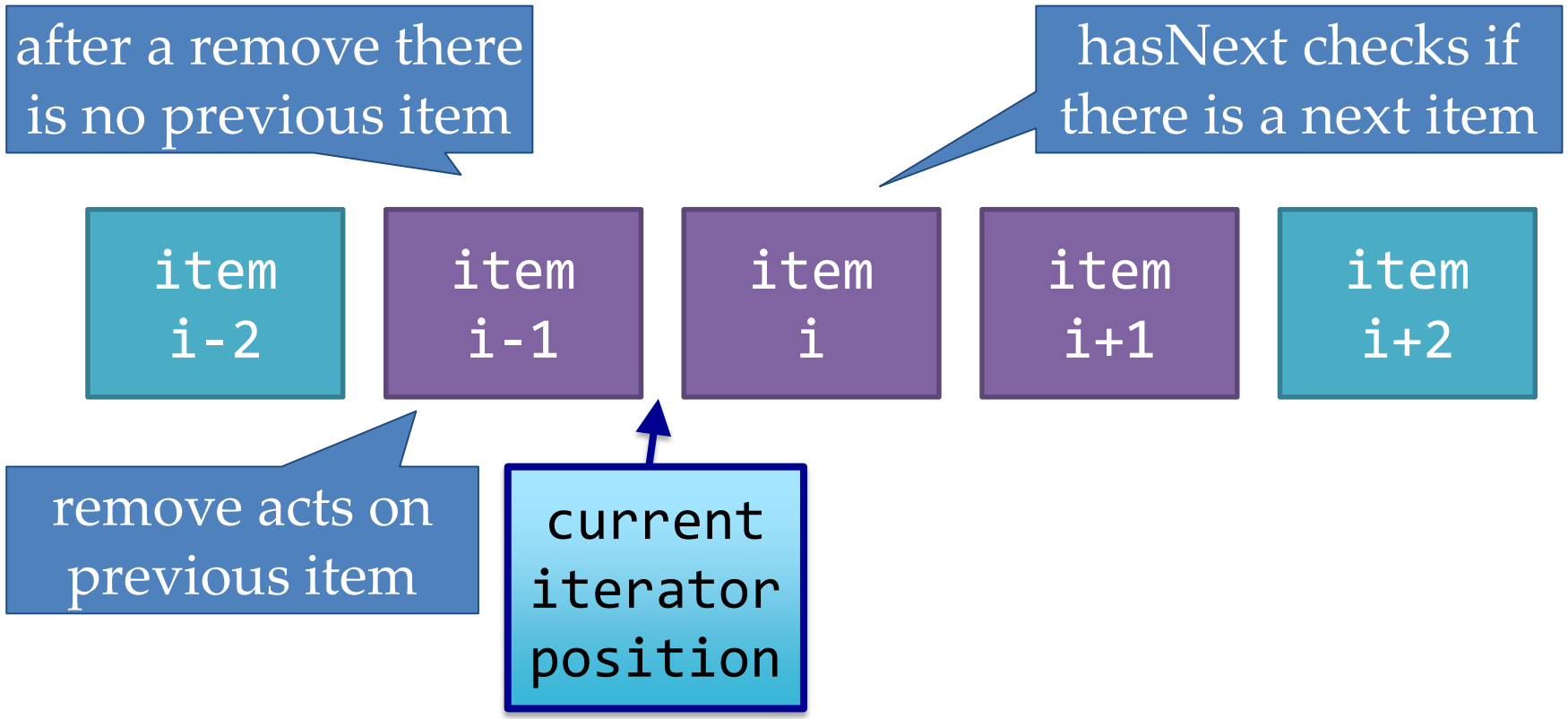
`Iterator` interface

an **`Iterator`** is conceptually *between* elements;
it does not refer to a particular object



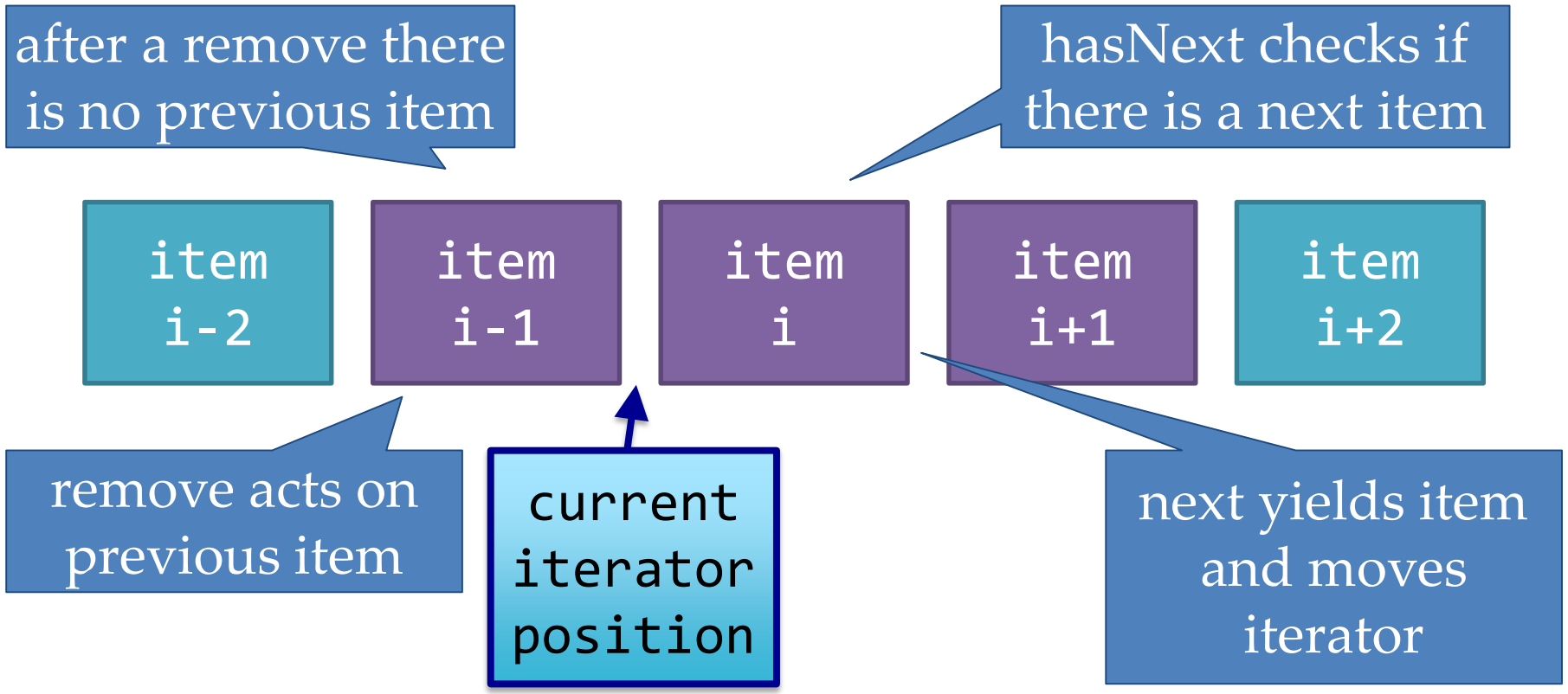
`Iterator` interface

an **`Iterator`** is conceptually *between* elements;
it does not refer to a particular object



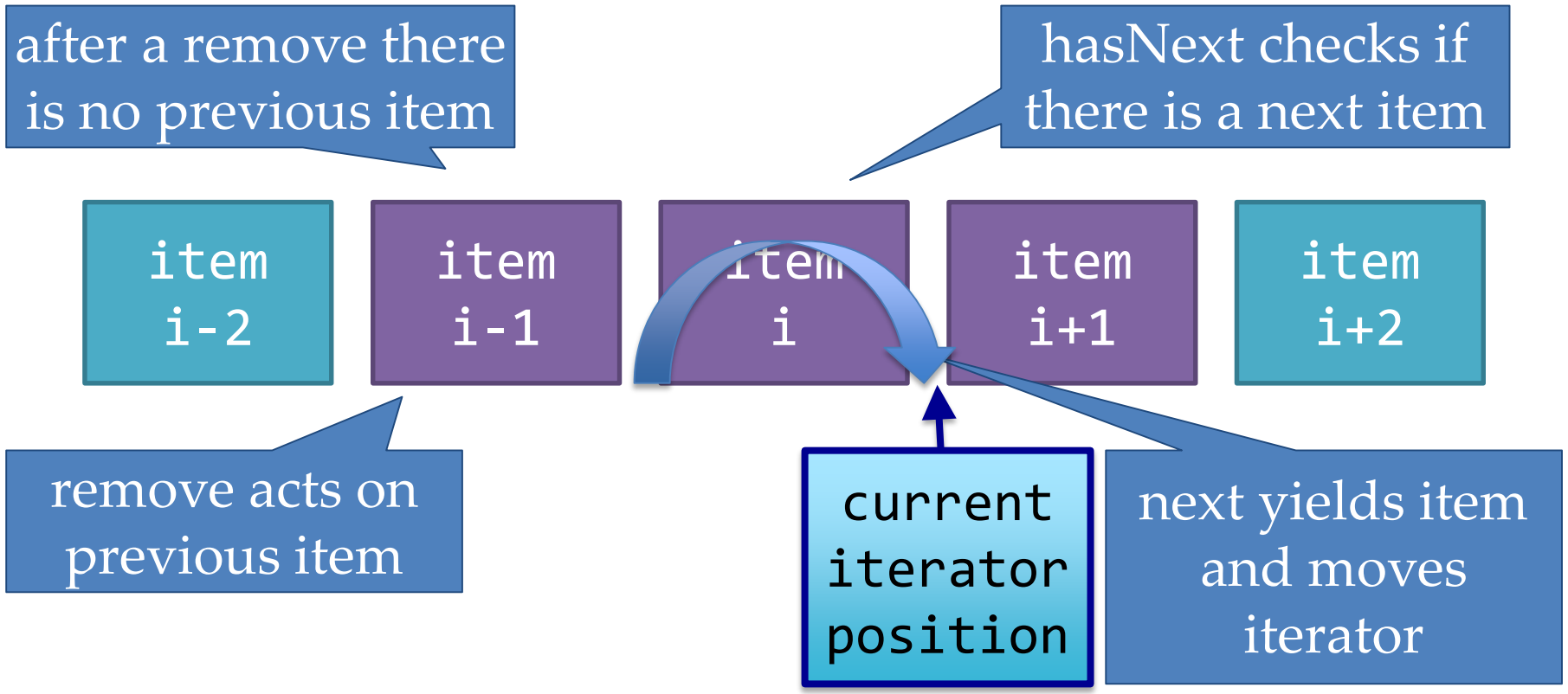
`Iterator` interface

an **`Iterator`** is conceptually *between* elements;
it does not refer to a particular object



`Iterator` interface

an **`Iterator`** is conceptually *between* elements;
it does not refer to a particular object



iterator applications in our Map: getting element

```
public V get(K key) {
  Iterator<Pair<K, V>> it = map.iterator();
  while (it.hasNext()) {
    Pair<K, V> pair = it.next();
    if (pair.getKey().equals(key)) {
      return pair.getVal();
    }
  }
  return null;
}
```

iterator applications in our Map: getting element

```
public V get(K key) {
  Iterator<Pair<K, V>> it = map.iterator();
  while (it.hasNext()) {
    Pair<K, V> pair = it.next();
    if (pair.getKey().equals(key)) {
      return pair.getVal();
    }
  }
  return null;
}
```

note how to get an iterator for `map`

iterator applications in our Map: getting element

```
public V get(K key) {
  Iterator<Pair<K, V>> it = map.iterator();
  while (it.hasNext()) {
    Pair<K, V> pair = it.next();
    if (pair.getKey().equals(key)) {
      return pair.getVal();
    }
  }
  return null;
}
```

note how to get an iterator for `map`

note: `next` yields the element **and** moves the iterator!

iterator applications in our Map: getting element

```
public V get(K key) {
  Iterator<Pair<K, V>> it = map.iterator();
  while (it.hasNext()) {
    Pair<K, V> pair = it.next();
    if (pair.getKey().equals(key)) {
      return pair.getVal();
    }
  }
  return null;
}
```

note how to get an iterator for `map`

note: `next` yields the element **and** moves the iterator!

using `next()` twice gives two elements

The most frequent error with iterators

iterator applications in our Map: removing element

```
public boolean remove (K key) {
  Iterator<Pair<K, V>> it = map.iterator();
  while (it.hasNext())
    if (it.next().getKey().equals(key)) {
      it.remove();
      return true;
  }
  return false;
}
```

iterator applications in our Map: removing element

```
public boolean remove (K key) {
  Iterator<Pair<K, V>> it = map.iterator();
  while (it.hasNext())
    if (it.next().getKey().equals(key)) {
      it.remove();
      return true;
  }
  return false;
}
```

single operation on list element; no need to keep it around in a variable

iterator applications in our Map: removing element

```
public boolean remove (K key) {
  Iterator<Pair<K, V>> it = map.iterator();
  while (it.hasNext())
    if (it.next().getKey().equals(key)) {
      it.remove();
      return true;
  }
  return false;
}
```

single operation on list element; no need to keep it around in a variable

this cannot be done by an enhanced for-loop because the Iterator is hidden!

be careful using multiple `Iterators` for the same `Iterable`

iterators are very handy to scan collections

- list, set, queue, ..

be careful using multiple `Iterators` for the same `Iterable`

iterators are very handy to scan collections

- list, set, queue, ..

what happens when the collection changes while the iterators scans it?

- `hasNext()` yields `true`, but the element is removed
- `hasNext()` yields `false`, but an element is inserted
- the previous element is removed or replaced, `remove()` fails

be careful using multiple `Iterators` for the same `Iterable`

iterators are very handy to scan collections

- list, set, queue, ..

what happens when the collection changes while the iterators scans it?

- `hasNext()` yields `true`, but the element is removed
- `hasNext()` yields `false`, but an element is inserted
- the previous element is removed or replaced, `remove()` fails

a `ConcurrentModificationException` might be thrown

- always when changes might effect the iterator

be careful using multiple `Iterators` for the same `Iterable`

iterators are very handy to scan collections

- list, set, queue, ..

what happens when the collection changes while the iterators scans it?

- `hasNext()` yields `true`, but the element is removed
- `hasNext()` yields `false`, but an element is inserted
- the previous element is removed or replaced, `remove()` fails

a `ConcurrentModificationException` might be thrown

- always when changes might effect the iterator

`next()` throws an `NoSuchElementException` when there is no element during normal iteration

be careful using multiple `Iterators` for the same `Iterable`

iterators are very handy to scan collections

- list, set, queue, ..

what happens when the collection changes while the iterators scans it?

- `hasNext()` yields `true`, but the element is removed
- `hasNext()` yields `false`, but an element is inserted
- the previous element is removed or replaced, `remove()` fails

a `ConcurrentModificationException` might be thrown

- always when changes might effect the iterator

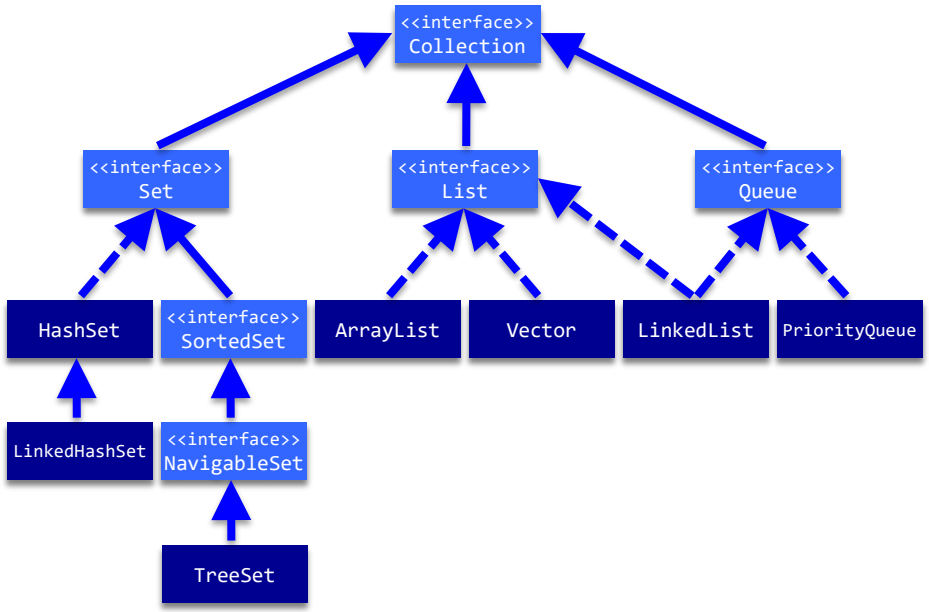
`next()` throws an `NoSuchElementException` when there is no element during normal iteration

`remove()` throws an `IllegalStateException` when there is nothing to be removed

Lists

interface **`List`** is an extension of **`Collection`**

- `List` adds methods to manipulate elements via indices
  - `void add(int index, E element)`
  - `E get(int index)`
  - `E remove (int index)`
  - `E set (int index, E element)` …



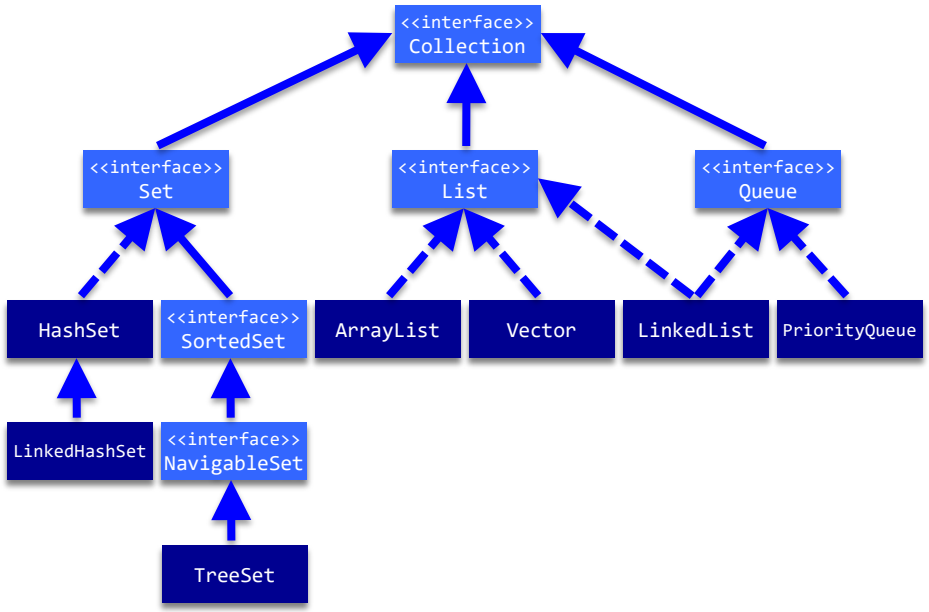
Lists

interface **`List`** is an extension of **`Collection`**

- `List` adds methods to manipulate elements via indices
  - **`void`** `add(`**`int`**` index, E element)`
  - `E get(`**`int`**` index)`
  - `E remove (`**`int`**` index)`
  - `E set (`**`int`**` index, E element) …`

`ArrayList` implements the interface **`List`**

other implementations are
`LinkedList` and **`Vector`**



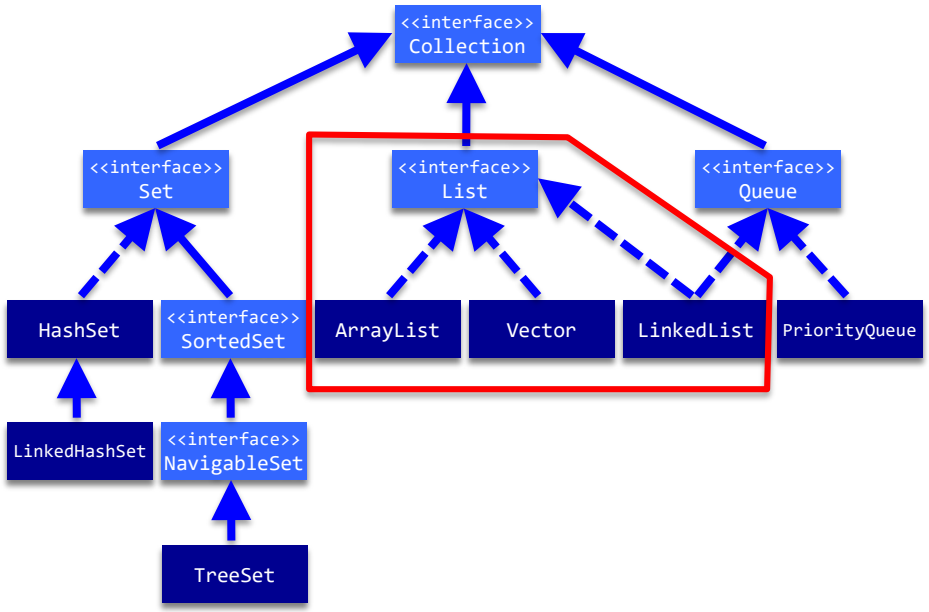
Lists

interface **`List`** is an extension of **`Collection`**

- `List` adds methods to manipulate elements via indices
  - **`void`** `add(`**`int`**` index, E element)`
  - `E get(`**`int`**` index)`
  - `E remove (`**`int`**` index)`
  - `E set (`**`int`**` index, E element)` …

`ArrayList` implements the interface **`List`**

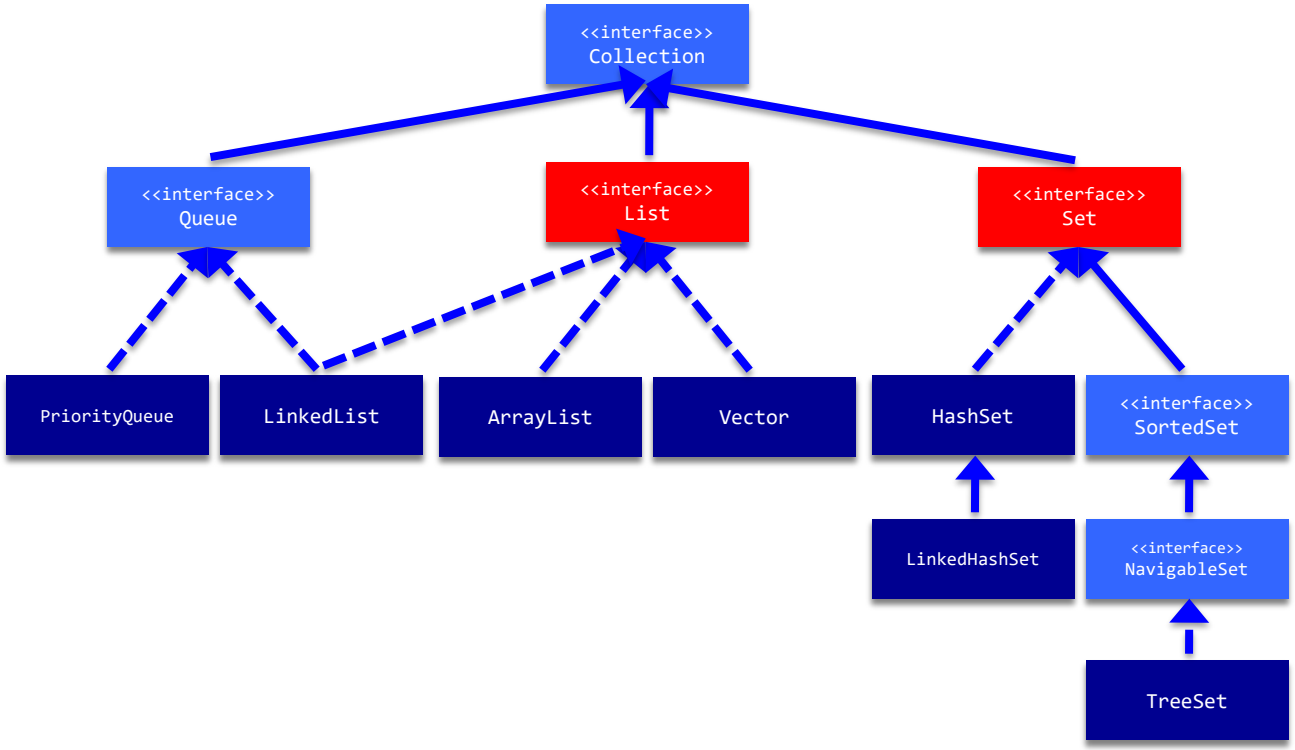
other implementations are
`LinkedList` and **`Vector`**



collection relationships

Set

- does not contain duplicates
- can (sometimes) be sorted !



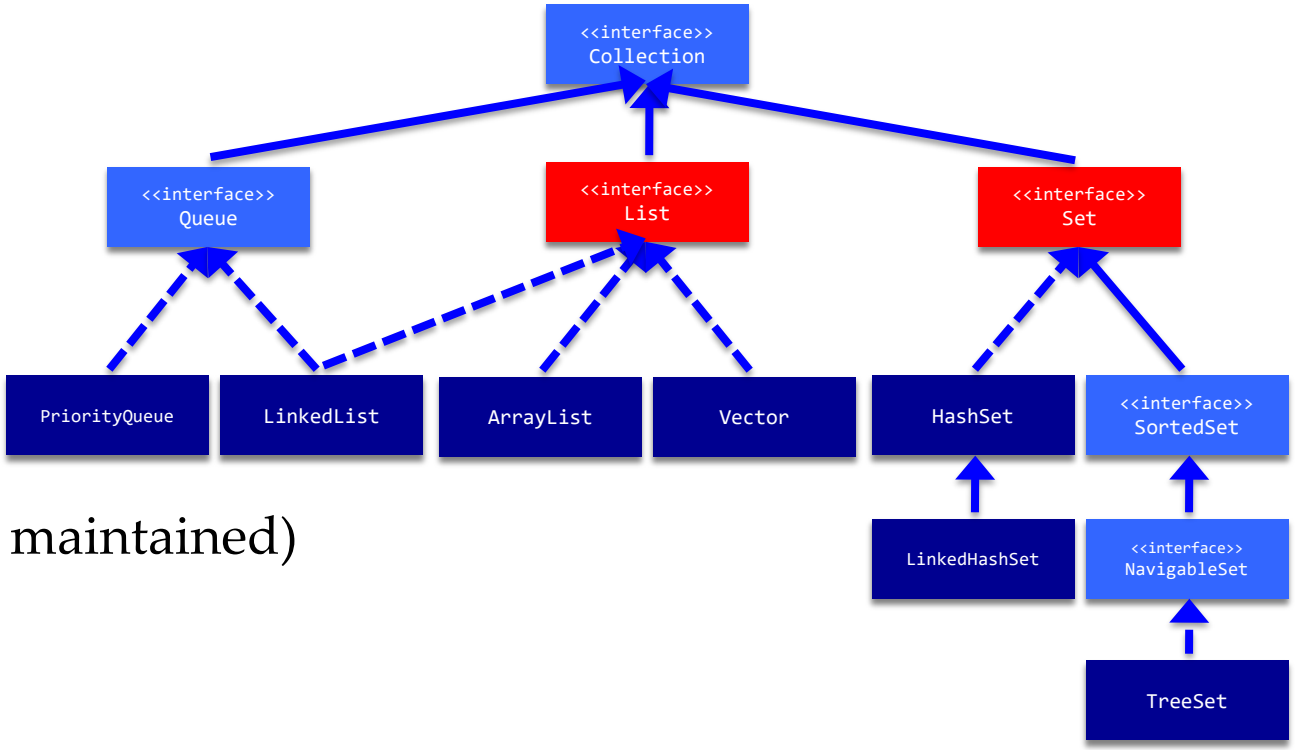
collection relationships

`Set`

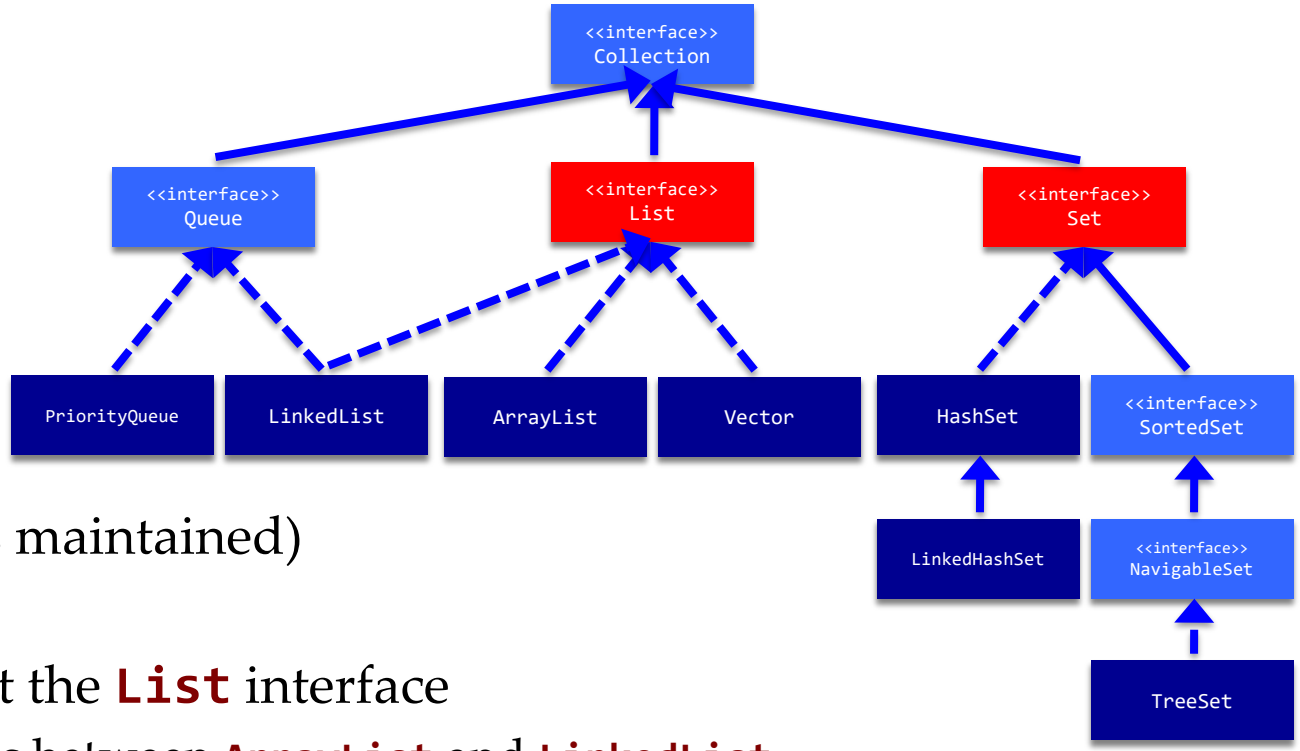
- does not contain duplicates
- can (sometimes) be sorted !

`List`

- elements are ordered (insertion order is maintained)
- elements can occur more than once



collection relationships



`Set`

- does not contain duplicates
- can (sometimes) be sorted !

`List`

- elements are ordered (insertion order is maintained)
- elements can occur more than once
- `ArrayList` and `LinkedList` implement the `List` interface
  - next lecture we will discuss the differences between `ArrayList` and `LinkedList`
- `Vector` is very similar to `ArrayList` in API,
  vectors are thread-safe and hence somewhat slower

the class `Collections`

do not confuse it with the interface `Collection`

- contains algorithms for collections
- like `Arrays` for arrays

implemented algorithms:

the class `Collections`

do not confuse it with the interface `Collection`

- contains algorithms for collections
- like `Arrays` for arrays

implemented algorithms:

```
sort, binarySearch, reverse, shuffle,
```

the class `Collections`

do not confuse it with the interface `Collection`

- contains algorithms for collections
- like `Arrays` for arrays

implemented algorithms:

```
sort, binarySearch, reverse, shuffle,
fill, copy, min, max, addAll,
```

the class `Collections`

do not confuse it with the interface `Collection`

- contains algorithms for collections
- like `Arrays` for arrays

implemented algorithms:

```
sort, binarySearch, reverse, shuffle,
fill, copy, min, max, addAll,
frequency, disjoint
```

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {



}
```

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);



}
```

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
```

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
```

+ any order possible
− get(i) can be inefficient

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
        for (Card card : list) {
}


        }
```

+ any order possible
− get(i) can be inefficient

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
        for (Card card : list) {
            if (card.face == Card.Face.Queen) {
                System.out.println("Queen2: " + card);
            }
        }
```

+ any order possible
− get(i) can be inefficient

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
```

+ any order possible
− get(i) can be inefficient

```
for (Card card : list) {
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen2: " + card);
    }
}
```

+ compact
+ efficient
− list cannot be changed

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
```

+ any order possible
− get(i) can be inefficient

```
for (Card card : list) {
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen2: " + card);
    }
}
```

+ compact
+ efficient
− list cannot be changed

```
Iterator<Card> iter = list.iterator();
while (iter.hasNext()) {

}
```

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
```

+ any order possible
− get(i) can be inefficient

```
for (Card card : list) {
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen2: " + card);
    }
}
```

+ compact
+ efficient
− list cannot be changed

```
Iterator<Card> iter = list.iterator();
while (iter.hasNext()) {
    Card card = iter.next();

}
```

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
```

+ any order possible
− get(i) can be inefficient

```
for (Card card : list) {
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen2: " + card);
    }
}
```

+ compact
+ efficient
− list cannot be changed

```
Iterator<Card> iter = list.iterator();
while (iter.hasNext()) {
    Card card = iter.next();
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen3: " + card);
    }
}
```

three ways to access all list elements

```
for (int i = 0; i < list.size(); i += 1) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
```

+ any order possible
− get(i) can be inefficient

```
for (Card card : list) {
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen2: " + card);
    }
}
```

+ compact
+ efficient
− list cannot be changed

```
Iterator<Card> iter = list.iterator();
while (iter.hasNext()) {
    Card card = iter.next();
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen3: " + card);
    }
}
```

+ efficient
+ flexible
+ not restricted to loop
! remove only last item

lessons learned

generics written as <T>

- make classes more general
- improves static typing

lessons learned

generics written as `<T>`

- make classes more general
- improves static typing

`Collection`: a family of generic data structures

- **`List`**
  - ordered like arrays

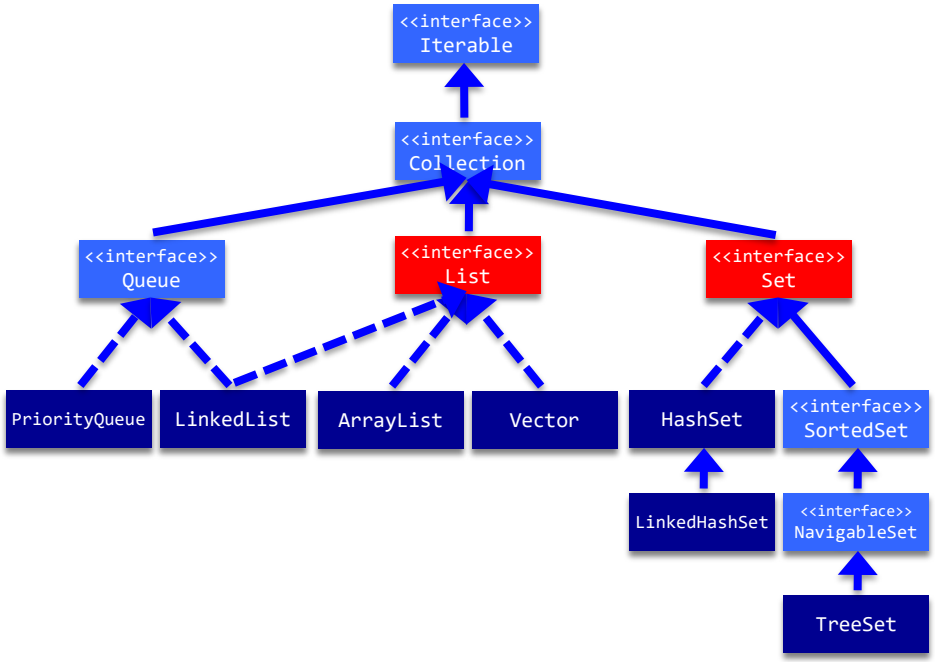
lessons learned

generics written as `<T>`

- make classes more general
- improves static typing

`Collection`: a family of generic data structures

- **`List`**
  - ordered like arrays
  - we can always add, insert or delete elements
- **`Set, Queue,`** ..



- **`Iterator`** is the standard way to access collections
  - `hasNext()`, `next()`, `remove()`

- **`Collections`** & **`Arrays`**: classes with convenience methods

Lecture 6: Generic & Recursive Data Structures