Advanced Object-oriented Programming

Lecture 31

Fun with Generics
A class can have more than one type parameter. For example, a class of pairs:

```java
public class Pair<A, B> {
    private A first;
    private B second;

    public Pair(A a, B b) {
        first = a;
        second = b;
    }
}
```
class Pair, contd.

```java
public A getFirst() {
    return first;
}

public B getSecond() {
    return second;
}
```

The constructor is declared without the `<__>` notation; however, actual parameters ‘must’ (= ‘should’) be supplied when the constructor is called.
Calling the Constructor

For example,

```java
public static void main(String[] args) {
    Integer i = new Integer(8);
    Pair<Integer, Integer> p = new Pair(i, i);
}
```

gives the compile-time warning (not error!):

```
Note: Pair.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```
Calling the Constructor

For example,

```java
public static void main(String[] args) {
    Integer i = new Integer(8);
    Pair<Integer, Integer> p = new Pair(i, i);
}
```

gives the compile-time warning (not error!):

```
Note: Pair.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```
javac -Xlint:unchecked
Pair.java

terminal output

Pair.java:37:  warning:  [unchecked] unchecked call to Pair(A,B) as a member of the raw type Pair
   Pair<Integer,Integer> p = new Pair(i,i);

so we ‘need’ (not desperately, as it does compile):

Pair<Integer,Integer> p =
   new Pair<Integer,Integer>(i,i);
javac -Xlint:unchecked
Pair.java

terminal output
Pair.java:37: warning: [unchecked] unchecked call to Pair(A,B) as a member of the raw type Pair
   Pair<Integer,Integer> p = new Pair(i,i);
so we ‘need’ (not desperately, as it does compile):

a better call
Pair<Integer,Integer> p =
   new Pair<Integer,Integer>(i,i);
Type Parameters

Within the class declaration, the formal parameter types act just like any other type, and can be used as types of parameters to methods and return types.

They can also be used as ‘actual’ parameter types:

```java
public class Pair<A, B> {
    ...
    public Pair<B, A> swap() {
        return new Pair<B, A>(second, first);
    }
}
```
More Types of Type Parameters

Consider a class that lays out GUI components one above another in a column. This might be used for several different uses, e.g.,

- setting menu items in a column
- setting Topic summaries (TopicIcons) in a column
- setting Messages (MessageIcons) in a column, etc.

In each of these example applications, the list of components to be formatted is homogeneous: all TopicIcons, or all MessageIcons, etc.

One way to do this would be to copy and paste the code from TopicIcon.java to the file MessageIcon.java, and do a global search/replace to replace all occurrences of ‘Topic’ with ‘Message’ (not very nice!)
More Types of Type Parameters

Consider a class that lays out GUI components one above another in a column. This might be used for several different uses, e.g.,

- setting menu items in a column
- setting Topic summaries (TopicIcons) in a column
- setting Messages (MessageIcons) in a column, etc.

In each of these example applications, the list of components to be formatted is homogeneous: all TopicIcons, or all MessageIcons, etc.

One way to do this would be to copy and paste the code from TopicIcon.java to the file MessageIcon.java, and do a global search/replace to replace all occurrences of ‘Topic’ with ‘Message’

(not very nice!)
More Types of Type Parameters

Consider a class that lays out GUI components one above another in a column. This might be used for several different uses, e.g.,

- setting menu items in a column
- setting Topic summaries (TopicIcons) in a column
- setting Messages (MessageIcons) in a column, etc.

In each of these example applications, the list of components to be formatted is homogeneous: all TopicIcons, or all MessageIcons, etc.

One way to do this would be to copy and paste the code from TopicIcon.java to the file MessageIcon.java, and do a global search/replace to replace all occurrences of ‘Topic’ with ‘Message’
(not very nice!)
A Question

Could we use generic classes to avoid having to copy and paste code?

```java
public class IconView<
C>
    extends JPanel {

    private C[] components;

    public void setComponents(C[] comps) {
        ...
        add(components[i]);
    }
}
```

C has to be a GUI component, i.e., a subclass of java.awt.Component.
A Question

Could we use generic classes to avoid having to copy and paste code?

```java
public class IconView<C> extends JPanel {
    private C[] components;
    public void setComponents(C[] comps) {
        ...
        add(components[i]);
    }
}
```

C has to be a GUI component, i.e., a subclass of java.awt.Component.
We can do it, but we need an extension...

```java
public class IconView<C extends Component>
    extends JPanel {

    private C[] components;

    public void setComponents(C[] comps) {
        components = comps;
        ...
        add(components[i]);
    }
}
```

Any **actual** parameter must be a subclass of java.awt.Component
We can do it, but we need an extension...

```java
public class IconView<C extends Component>
   extends JPanel {

   private C[] components;

   public void setComponents(C[] comps) {
      components = comps;
      ...
      add(components[i]);
   }
}
```

Any **actual** parameter must be a subclass of java.awt.Component
Constructors Again

IconView\(<\text{TopicIcon}\>\) topicView =
    new IconView\(<\text{TopicIcon}\>\)();
TopicIcon\[] topics = ...;
topicView.setComponents(topics);
...
IconView\(<\text{MessageIcon}\>\) messageView =
    new IconView\(<\text{MessageIcon}\>\)();
MessageIcon\[] messages = ...;
messageView.setComponents(messages);
Formal parameter `C` is replaced by actual parameter `TopicIcon`. The compiler checks:

- `TopicIcon` is a subclass of `Component`
- `topics` is of type `TopicIcon[]`

```java
public class IconView<C extends Component> extends JPanel {
    ...

    public void setComponents(C[] comps) {
        ...
    }
}
```
Type Checking Again

IconView<TopicIcon> topicView =
    new IconView<TopicIcon>();
...
topicView.setComponents(topics);

Formal parameter C is replaced by actual parameter TopicIcon. The compiler checks:

- TopicIcon is a subclass of Component
- topics is of type TopicIcon[]

public class IconView<C extends Component>
    extends JPanel {
    ...
    public void setComponents(C[] comps) {
        ...
    }
}
Type Checking Again

IconView<TopicIcon> topicView =
   new IconView<TopicIcon>();
...
topicView.setComponents(topics);

Formal parameter \( C \) is replaced by actual parameter \( TopicIcon \). The compiler checks:

- \( TopicIcon \) is a subclass of \( Component \)
- \( topics \) is of type \( TopicIcon[] \)

public class IconView<C extends Component>
   extends JPanel {
   ...
   public void setComponents(C[] comps) {
      ...
   }
}
IconView<TopicIcon> topicView =
    new IconView<TopicIcon>();
...
topicView.setComponents(topics);

Formal parameter \( C \) is replaced by actual parameter \( \text{TopicIcon} \). The compiler checks:

- \( \text{TopicIcon} \) is a subclass of \( \text{Component} \)
- \( \text{topics} \) is of type \( \text{TopicIcon}[] \)

```java
public class IconView<C extends Component> extends JPanel {
    ...
    public void setComponents(C[] comps) {
        ...
    }
}
```
IconView<TopicIcon> topicView =
    new IconView<TopicIcon>();
...
    topicView.setComponents(topics);

Formal parameter C is replaced by actual parameter TopicIcon. The compiler checks:

- TopicIcon is a subclass of Component
- topics is of type TopicIcon[

public class IconView<C extends Component>
    extends JPanel {
    ...
    public void setComponents(C[] comps) {
        ...
    }
}
Type Checking Again

IconView<TopicIcon> topicView =
    new IconView<TopicIcon>();
...
topicView.setComponents(topics);

Formal parameter \( c \) is replaced by actual parameter \( \text{TopicIcon} \). The compiler checks:

- \( \text{TopicIcon} \) is a subclass of \( \text{Component} \)
- \( \text{topics} \) is of type \( \text{TopicIcon[]} \)

public class IconView<T extends Component>
    extends JPanel {
    ...
    public void setComponents(T[] comps) {
        ...
    }
}
class Extreme<A, B extends A, C extends B> {

    Extreme(){

        public static void main(String[] args) {

            Extreme<Component, Container, JComponent> e =
            new Extreme<Component, Container, JComponent>();

        }

    }

    This is correct, because JComponent is a subclass of
    Container, which is a subclass of Component.
class Extreme<A, B extends A, C extends B> {
    Extreme(){}
    public static void main(String[] args) {
        Extreme<Component,Container,JComponent> e = new Extreme<Component,Container,JComponent>();
    }
}

This is correct, because JComponent is a subclass of Container, which is a subclass of Component.
class Extreme<A, B extends A, C extends B> {
    Extreme(){}
    public static void main(String[] args) {
        Extreme<Component, Container, JComponent> e =
        new Extreme<Component, Container, JComponent>();
    }
}

This is correct, because JComponent is a subclass of Container, which is a subclass of Component.
More and More

class Extreme<A, B extends A, C extends B> {
    Extreme(){}
    public static void main(String[] args) {
        Extreme<Component,Container,JComponent> e = new Extreme<Component,Container,JComponent>();
    }
}

This is correct, because JComponent is a subclass of Container, which is a subclass of Component.
class Extreme<A, B extends A, C extends B> {
    Extreme(){}

    public static void main(String[] args) {
        Extreme<Component, Container, JComponent> e = new Extreme<Component, Container, JComponent>();
    }
}

This is correct, because JComponent is a subclass of Container, which is a subclass of Component.
More and More

class Extreme\<A, B extends A, C extends B\> {  
    Extreme(){}  
    public static void main(String[] args) {  
        Extreme\<Component, Container, JComponent\> e = new Extreme\<Component, Container, JComponent\>();  
    }  
}  

This is correct, because \texttt{JComponent} is a subclass of \texttt{Container}, which is a subclass of \texttt{Component}.
class Extreme<A, B extends A, C extends B> {
    Extreme(){
    }
    public static void main(String[] args) {
        Extreme<Component,Container,JComponent> e =
        new Extreme<Component,Container,JComponent>();
    }
}

This is correct, because JComponent is a subclass of Container, which is a subclass of Component.
class Extreme<A, B extends A, C extends B> {
    Extreme(){}

    public static void main(String[] args) {
        Extreme<Component, Container, JComponent> e = new Extreme<Component, Container, JComponent>();
    }
}

This is correct, because **JComponent** is a subclass of **Container**, which is a subclass of **Component**.
More and More

```java
class Extreme<A, B extends A, C extends B> {
    Extreme(){}

    public static void main(String[] args) {
        Extreme<Component,Container,JComponent> e =
        new Extreme<Component,Container,JComponent>();
    }
}
```

This is correct, because `JComponent` is a subclass of `Container`, which is a subclass of `Component`. 
class Extreme<A, B extends A, C extends B> {
    Extreme(){}

    public static void main(String[] args) {
        Extreme<Component,Container,JComponent> e = new Extreme<Component,Container,JComponent>();
    }
}

This is correct, because JComponent is a subclass of Container, which is a subclass of Component.
The keyword `extends` also applies to interfaces. For example:

```java
public class RunnablePool<A extends Runnable> {
    private A[] pool;
    ...
}
```

This can be instantiated with any class that implements `Runnable`, e.g.,

```java
RunnablePool<Consumer> consumerPool = new RunnablePool<Consumer>;
```
The keyword `extends` also applies to interfaces. For example

```java
public class RunnablePool<A extends Runnable> {
    private A[] pool;
    ...
}
```

This can be instantiated with any class that implements `Runnable`, e.g.,

```java
RunnablePool<Consumer> consumerPool = new RunnablePool<Consumer>;
RunnablePool<EchoHandler> echoPool = new RunnablePool<EchoHandler>;
```
The keyword `extends` also applies to `interfaces`. For example

```java
public class RunnablePool<A extends Runnable> {
    private A[] pool;
    ...
}
```

This can be instantiated with any class that implements `Runnable`, e.g.,

```java
RunnablePool<Consumer> consumerPool =
    new RunnablePool<Consumer>;
RunnablePool<EchoHandler> echoPool =
    new RunnablePool<EchoHandler>;
```
The keyword `extends` also applies to interfaces. For example

```java
public class RunnablePool<A extends Runnable> {
    private A[] pool;
    ...
}
```

This can be instantiated with any class that implements `Runnable`, e.g.,

```java
RunnablePool<Consumer> consumerPool = new RunnablePool<Consumer>;
RunnablePool<EchoHandler> echoPool = new RunnablePool<EchoHandler>;
```
The keyword `extends` also applies to interfaces. For example

```java
public class RunnablePool<A extends Runnable> {
    private A[] pool;
    ...
}
```

This can be instantiated with any class that implements `Runnable`, e.g.,

```java
RunnablePool<Consumer> consumerPool = new RunnablePool<Consumer>;
RunnablePool<EchoHandler> echoPool = new RunnablePool<EchoHandler>;
```
The keyword `extends` also applies to interfaces. For example:

```java
public class RunnablePool<A extends Runnable> {
    private A[] pool;
    ...
}
```

This can be instantiated with any class that implements `Runnable`, e.g.,

```java
RunnablePool<Consumer> consumerPool =
    new RunnablePool<Consumer>;
```

```java
RunnablePool<EchoHandler> echoPool =
    new RunnablePool<EchoHandler>;
```
The keyword extends also applies to interfaces. For example:

```java
public class RunnablePool<A extends Runnable> {
    private A[] pool;
    ...
}
```

This can be instantiated with any class that implements Runnable, e.g.,

```java
RunnablePool<Consumer> consumerPool = new RunnablePool<Consumer>;
RunnablePool<EchoHandler> echoPool = new RunnablePool<EchoHandler>;
```
Generic Interfaces

Interfaces can themselves have type parameters. For example

```java
interface Listable<A> {
    public Vector<A> list();
}
```

Classes implement this by providing a method called `list` that returns a `Vector` of some type `A`. E.g.,

```java
class AVLTree implements Listable<Integer> {
    public Vector<Integer> list() {
        ...
    }
    ...
}
```
Generic Interfaces

Interfaces can themselves have type parameters. For example

```java
interface Listable<A> {
    public Vector<A> list();
}
```

Classes implement this by providing a method called `list` that returns a `Vector` of some type `A`. E.g.,

```java
class AVLTree implements Listable<Integer> {
    public Vector<Integer> list() {
        ...
    }
    ...
}
```
Interfaces can themselves have type parameters. For example

```java
interface Listable<A> {
    public Vector<A> list();
}
```

Classes implement this by providing a method called `list` that returns a Vector of some type `A`. E.g.,

```java
class AVLTree implements Listable<Integer> {
    public Vector<Integer> list() {
        ...
    }
    ...
    ...
}
```
Generic Interfaces

Interfaces can themselves have type parameters. For example

```java
interface Listable<A> {
    public Vector<A> list();
}
```

Classes implement this by providing a method called `list` that returns a `Vector` of some type `A`. E.g.,

```java
class AVLTree implements Listable<Integer> {
    public Vector<Integer> list() {
        ...
        ...
    }
}
```
Interfaces can themselves have type parameters. For example

```java
interface Listable<A> {
    public Vector<A> list();
}
```

Classes implement this by providing a method called `list` that returns a `Vector` of some type `A`. E.g.,

```java
class AVLTree implements Listable<Integer> {
    public Vector<Integer> list() {
        ...
    }
    ...
}
```
Generic Interfaces

Interfaces can themselves have type parameters. For example

```java
interface Listable<A> {
    public Vector<A> list();
}
```

Classes implement this by providing a method called `list` that returns a `Vector` of some type `A`. E.g.,

```java
class AVLTree implements Listable<Integer> {
    public Vector<Integer> list() {
        ...
    }
    ...
}
```
Generic Interfaces

Interfaces can themselves have type parameters. For example

```java
interface Listable<A> {
    public Vector<A> list();
}
```

Classes implement this by providing a method called `list` that returns a `Vector` of some type `A`. E.g.,

```java
class AVLTree implements Listable<Integer> {
    public Vector<Integer> list() {
        ...
    }
    ...
}
```
Interfaces can themselves have type parameters. For example

```java
interface Listable<A> {
    public Vector<A> list();
}
```

Classes implement this by providing a method called `list` that returns a `Vector` of some type `A`. E.g.,

```java
class AVLTree implements Listable<Integer> {
    public Vector<Integer> list() {
        ...
    }
    ...
}
```
Exercise!

Implement this. Here’s a hint:

```java
/**
 * Return a vector containing all the elements
 * in the tree.
 * @return the <b>sorted</b> list of all the
 *         elements in the tree
 */
public Vector<Integer> list() {
    Vector<Integer> v = new Vector<Integer>();
    if (root != null) {
        v = root.list(v);
    }
    return v;
}
```
Exercise!

in class AVLTreeNode

```java
/**
 * <em>Add</em> all the elements in this node to the given vector.
 * @param the vector to add the values to
 * @return the vector containing all the elements in v, followed by all the elements in the left subtree, followed by the internal label, followed by all the elements in the right subtree
 */
Vector<Integer> list(Vector<Integer> v) {
    ...
}
```

Final hint: look at AVLTree#printInorder().
Generic AVL Trees

The AVL trees of the last couple of lectures stored integers in balanced binary search trees.

This efficient data structure is useful for storing other data types; for example,

- Strings (recently used email addresses, perhaps)
- Topics in a Message Board forum (sorted by integer ID, perhaps)
- Employee records (sorted by Payroll number, or National Insurance number)

So it would be useful to have a generic class `AVLTree<A>`.

```java
public class AVLTree<A> { ... }
```
Generic AVL Trees

The AVL trees of the last couple of lectures stored integers in balanced binary search trees.

This efficient data structure is useful for storing other data types; for example,

- Strings (recently used email addresses, perhaps)
- Topics in a Message Board forum (sorted by integer ID, perhaps)
- Employee records (sorted by Payroll number, or National Insurance number)

So it would be useful to have a generic class `AVLTree<A>`.

```java
public class AVLTree<A> { ... }
```
Generic AVL Trees

The AVL trees of the last couple of lectures stored integers in balanced binary search trees.

This efficient data structure is useful for storing other data types; for example,

- Strings (recently used email addresses, perhaps)
- Topics in a Message Board forum (sorted by integer ID, perhaps)
- Employee records (sorted by Payroll number, or National Insurance number)

So it would be useful to have a generic class `AVLTree<A>`.

```java
public class AVLTree<A> { ... }
```
Generic AVL Trees

The AVL trees of the last couple of lectures stored integers in balanced binary search trees.

This efficient data structure is useful for storing other data types; for example,

- Strings (recently used email addresses, perhaps)
- Topics in a Message Board forum (sorted by integer ID, perhaps)
- Employee records (sorted by Payroll number, or National Insurance number)

So it would be useful to have a generic class `AVLTTree<A>`. ```
public class AVLTTree<A> { ... }
```
Generic AVL Trees

The AVL trees of the last couple of lectures stored integers in balanced binary search trees.

This efficient data structure is useful for storing other data types; for example,

- Strings (recently used email addresses, perhaps)
- Topics in a Message Board forum (sorted by integer ID, perhaps)
- Employee records (sorted by Payroll number, or National Insurance number)

So it would be useful to have a generic class `AVLTree<A>`.

```java
public class AVLTree<A> { ... }
```
Generic AVL Trees

The AVL trees of the last couple of lectures stored integers in balanced binary search trees.

This efficient data structure is useful for storing other data types; for example,

- Strings (recently used email addresses, perhaps)
- Topics in a Message Board forum (sorted by integer ID, perhaps)
- Employee records (sorted by Payroll number, or National Insurance number)

So it would be useful to have a generic class `AVLTree<A>`.

```java
public class AVLTree<A> { ... }
```
Consider the look-up method `isIn()`:

```java
public boolean isIn(A v) {
    if (v.equals(value)) {
        return true;
    } else if (v < value) {
        ...
    }
}
```

The parameter is some object belonging to the generic (formal parameter) class `A`. Because `v` is an object — an instance of a class — we use `equals()` to test equality. Oops. We can’t do this: less-than only works for numbers.
Generic AVLTrees

Consider the look-up method `isIn()`:

```java
public boolean isIn(A v) {
    if (v.equals(value)) {
        return true;
    } else if (v < value) {
        ...
    }
}
```

The parameter is some object belonging to the generic (formal parameter) class `A`.
Because `v` is an object — an instance of a class — we use `equals()` to test equality.
Oops. We can’t do this: less-than only works for numbers.
Consider the look-up method `isIn()`:

```java
public boolean isIn(A v) {
    if (v.equals(value)) {
        return true;
    } else if (v < value) {
        ...
    }
}
```

The parameter is some object belonging to the generic (formal parameter) class `A`. Because `v` is an object — an instance of a class — we use `equals()` to test equality. Oops. We can’t do this: less-than only works for numbers.
Consider the look-up method `isIn()`:

```java
public boolean isIn(A v) {
    if (v.equals(value)) {
        return true;
    } else if (v < value) {
        ...
    }
}
```

The parameter is some object belonging to the generic (formal parameter) class `A`. Because `v` is an object — an instance of a class — we use `equals()` to test equality. Oops. We can’t do this: less-than only works for numbers.
Generic AVLTrees

Consider the look-up method `isIn()`:

```java
public boolean isIn(A v) {
    if (v.equals(value)) {
        return true;
    } else if (v < value) {
        ...
    }
}
```

The parameter is some object belonging to the generic (formal parameter) class `A`. Because `v` is an object — an instance of a class — we use `equals()` to test equality.

Oops. We can’t do this: less-than only works for numbers.
Consider the look-up method `isIn()`:

```java
public boolean isIn(A v) {
    if (v.equals(value)) {
        return true;
    } else if (v < value) {
        ...
    }
}
```

The parameter is some object belonging to the generic (formal parameter) class `A`. Because `v` is an object — an instance of a class — we use `equals()` to test equality.

Oops. We can’t do this: less-than only works for numbers.
But we can compare objects.

For example, we can compare instances of class `String` by lexicographic (‘alphabetical’) order, or instances of class `Date` by chronological order, instances of List classes by prefix order (one list is less than another if the first is a prefix of the other), and so on.

You can easily imagine a class of Employee Records whose instances could be compared by lexicographic order of name, or by chronological order of Date of Birth, or by integer order of age or payroll number, etc., etc.
But we can compare objects.

For example, we can compare instances of class `String` by lexicographic (‘alphabetical’) order, or instances of class `Date` by chronological order, instances of List classes by prefix order (one list is less than another if the first is a prefix of the other), and so on.

You can easily imagine a class of Employee Records whose instances could be compared by lexicographic order of name, or by chronological order of Date of Birth, or by integer order of age or payroll number, etc., etc.
We would like to say that the internal labels of an AVL Tree (or a Binary Search Tree) can belong to any class that has some order on its instances.

For example, Strings with lexicographic order, Dates with chronological order, etc.

... which sounds like we need an interface (specifying some method that compares two instances).
We would like to say that the internal labels of an AVL Tree (or a Binary Search Tree) can belong to any class that has some order on its instances.

For example, Strings with lexicographic order, Dates with chronological order, etc.

...which sounds like we need an interface (specifying some method that compares two instances).
Interface java.lang.Comparable

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

Implementing classes need to provide a method that will compare an instance of the class to an instance of some class \( T \).

- negative if the instance is less than the parameter;
- 0 if the instance is equal to the parameter;
- positive if the instance is greater than the parameter.

Any old class?
public interface Comparable<T> {
    public int compareTo(T o);
}

Implementing classes need to provide a method that will compare an instance of the class to an instance of some class T.

- negative if the instance is less than the parameter;
- 0 if the instance is equal to the parameter;
- positive if the instance is greater than the parameter.

Any old class?
Implementing classes need to provide a method that will compare an instance of the class to an instance of some class T.

- negative if the instance is less than the parameter;
- 0 if the instance is equal to the parameter;
- positive if the instance is greater than the parameter.

Any old class?
public interface Comparable<T> {
    public int compareTo(T o);
}

Implementing classes need to provide a method that will compare an instance of the class to an instance of some class T.

negative if the instance is less than the parameter;
0 if the instance is equal to the parameter;
positive if the instance is greater than the parameter.
Any old class?
public interface Comparable<T> {
    public int compareTo(T o);
}

Implementing classes need to provide a method that will compare an instance of the class to an instance of some class \( T \).

negative if the instance is less than the parameter;
0 if the instance is equal to the parameter;
positive if the instance is greater than the parameter.

Any old class?
Interface java.lang.Comparable

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

Implementing classes need to provide a method that will compare an instance of the class to an instance of some class T.

negative if the instance is less than the parameter;
0 if the instance is equal to the parameter;
positive if the instance is greater than the parameter.

Any old class?
Implementing Comparable<T>

Generally, implementing classes compare instances to instances of the *same* class:

```java
public class Integer
    implements Comparable<Integer> {
    ...
    public int compareTo(Integer o) {
        ...
    }
}
```
Implementing Comparable<T>

Generally, implementing classes compare instances to instances of the same class:

```java
public class Integer
    implements Comparable<Integer> { 
    ...
    public int compareTo(Integer o) { 
        ...
    }
}
```
Implementing Comparable<T>

Generally, implementing classes compare instances to instances of the *same* class:

```java
in package java.lang

public class Integer implements Comparable<Integer> {
    ...
    public int compareTo(Integer o) {
        ...
    }
}
```
Implementing Comparable<T>

```java
public class Date implements Comparable<Date> {
    ...
    public long getTime() {... }
    public int compareTo(Date d) {
        return (int)this.getTime() - d.getTime();
    }
}
```
class AVLTree< T extends Comparable<T> > {
    public boolean isIn(T v) {
        AVLTreeNode tn = root;
        if (v.compareTo(tn.value) == 0) {
            return true;
        } else if (v.compareTo(tn.value) < 0) {
            ...
        }
    }
}
class AVLTree\< T extends Comparable\<T>> {  
    public boolean isIn(T v) {  
        AVLTreeNode tn = root;  
        if (v.compareTo(tn.value) == 0) {  
            return true;  
        } else if (v.compareTo(tn.value) < 0) {  
            ...  
        }  
    }  
}
class AVLTree<T extends Comparable<T>> {
    public boolean isIn(T v) {
        AVLTreeNode tn = root;
        if (v.compareTo(tn.value) == 0) {
            return true;
        } else if (v.compareTo(tn.value) < 0) {
            ...
        }
    }
}
Another Example

An interface specifying ‘functions’ that take some input (of type A) and return some output (of type B):

```java
interface Function<A, B> {
    public B function(A a);
}
```

Implementing classes must have a method called `function` with one parameter (of any type) and some (non-void) return type.
An interface specifying ‘functions’ that take some input (of type \texttt{A}) and return some output (of type \texttt{B}):\[
\text{interface Function}\langle\texttt{A,B}\rangle\ {\{}
\begin{align*}
\text{public B function(}\texttt{A a});
\end{align*}
\}\]
Implementing classes must have a method called \texttt{function} with one parameter (of any type) and some (non-void) return type.
An interface specifying ‘functions’ that take some input (of type `A`) and return some output (of type `B`):

```java
interface Function<A, B> {
    public B function(A a);
}
```

Implementing classes must have a method called `function` with one parameter (of any type) and some (non-void) return type.
Another Example

An interface specifying ‘functions’ that take some input (of type A) and return some output (of type B):

```java
interface Function<A, B> {
    public B function(A a);
}
```

Implementing classes must have a method called `function` with one parameter (of any type) and some (non-void) return type.
Functions

We can specialise this to functions whose return type is the same as the parameter type:

```java
interface IterableFunction<A>
    extends Function<A, A> {

}
```

This is the same as

```java
interface IterableFunction<A> {
    public A function(A a);
}
```
Functions

We can specialise this to functions whose return type is the same as the parameter type:

```java
interface IterableFunction<A>
    extends Function<A,A> {

}
```

This is the same as

```java
interface IterableFunction<A> {
    public A function(A a);
}
```
Example

An example of an implementing class that takes an Integer as input and returns an Integer as output:

class DoubleInt
  implements IterableFunction<Integer> {

    public Integer function(Integer i) {
      return 2 * i;
    }
  }

Example

An example of an implementing class that takes an `Integer` as input and returns an `Integer` as output:

class DoubleInt
    implements IterableFunction<Integer> {

    public Integer function(Integer i) {
        return 2 * i;
    }

    }

A generic class implementing a generic interface.

This class will repeatedly call function() a fixed number of times.

class RepeatFunction<A>
  implements IterableFunction<A> {

  private int count;
  private IterableFunction<A> fun;

  RepeatFunction(int i, IterableFunction f) {
    count = i;
    fun = f;
  }
}
Another Example

```java
public A function(A a) {
    A val = a;
    int i = count;
    while (i-- > 0) {
        a = fun.function(a);
    }
    return val;
}
```
We multiply by eight by doubling then doubling then doubling then doubling:

```java
public static void main(String[] args) {
    DoubleInt doubler = new DoubleInt();
    RepeatFunction<Integer> rf =
        new RepeatFunction<Integer>(3, doubler);
    System.out.println(rf.function(5));
}
```

`rf` will double three times.
The result should be 40.
We multiply by eight by doubling then doubling then doubling:

```java
public static void main(String[] args) {
    DoubleInt doubler = new DoubleInt();
    RepeatFunction<Integer> rf =
        new RepeatFunction<Integer>(3, doubler);
    System.out.println(rf.function(5));
}
```

rf will double three times.
The result should be 40.
We multiply by eight by doubling then doubling then doubling:

```java
public static void main(String[] args) {
    DoubleInt doubler = new DoubleInt();
    RepeatFunction<Integer> rf =
        new RepeatFunction<Integer>(3, doubler);
    System.out.println(rf.function(5));
}
```

rf will double three times.
The result should be 40.
We multiply by eight by doubling then doubling then doubling:

```java
public static void main(String[] args) {
    DoubleInt doubler = new DoubleInt();
    RepeatFunction<Integer> rf =
        new RepeatFunction<Integer>(3, doubler);
    System.out.println(rf.function(5));
}
```

rf will double three times.
The result should be 40.
That’s All, Folks!

Summary

- Multiple Type Parameters
- extends
- Generic Interfaces

Next:

Generic methods