Chapter 18 - Generic Classes

Wildcard Types

Name	Syntax	Meaning
Wildcard with lower bound	`? extends B`	Any subtype of `B`
Wildcard with upper bound	`? super B`	Any supertype of `B`
Unbounded wildcard	`?`	Any type

Wildcard Types

- Wildcard types are used to formulate subtle constraints on type parameters.
- A wildcard type is a type that can remain unknown.
- A method in a `LinkedList` class to add all elements of `LinkedList other`:

`other` can be of any subclass of `E`.

```
public void addAll(LinkedList<? extends E> other)
{
   ListIterator<E> iter = other.listIterator();
   while (iter.hasNext())
   {
      add(iter.next());
   }
}
```

```
public void addAll(LinkedList<? super E> other)
{
   ListIterator<E> iter = other.listIterator();
   while (iter.hasNext())
   {
      add(iter.next());
   }
}
```

Wildcard Types

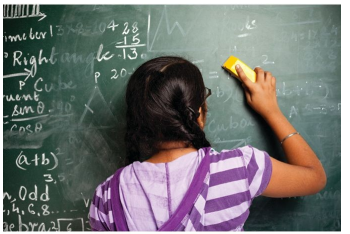
- A method in the `Collections` class which uses an unbounded wildcard:

```
static void reverse(List<?> list)
```

- You can think of that declaration as a shorthand for:

```
static void <T> reverse(List<T> list)
```

Type Erasure



In the Java virtual machine, generic types are erased.

Type Erasure

- The virtual machine erases type parameters, replacing them with their bounds or `Objects.`
- For example, generic class `Pair<T, S>` turns into the following raw class:

```
public class Pair
{
  private Object first;  private Object second;

  public Pair(Object firstElement, Object secondElement)
  {
    first = firstElement;  second = secondElement;
  }
  public Object getFirst() { return first; }
  public Object getSecond() { return second; }
}
```

Type Erasure

- Same process is applied to generic methods.
- In this generic method:

```
public static <E extends Measurable> E min(E[] objects)
{
  E smallest = objects[0];
  for (int i = 1; i < objects.length; i++)
  {
    E obj = objects[i];
    if (obj.getMeasure() < smallest.getMeasure())
    {
   smallest = obj;
    }
  }
  return smallest;
}
```

- The type parameter Is replaced with its bound `Measurable`:

```
public static Measurable min(Measurable[] objects)
{
   Measurable smallest = objects[0];
   for (int i = 1; i < objects.length; i++)
   {
      Measurable obj = objects[i];
      if (obj.getMeasure() < smallest.getMeasure())
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

Type Erasure

- Knowing about type erasure helps you understand limitations of Java generics.
- You cannot construct new objects of a generic type.
- For example, trying to fill an array with copies of default objects would be wrong:

```
public static <E> void fillWithDefaults(E[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new E(); // ERROR
}
```

- Type erasure yields:

```
public static void fillWithDefaults(Object[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new Object(); // Not useful
}
```

Type Erasure

- To solve this particular problem, you can supply a default object:

```
public static <E> void fillWithDefaults(E[] a, E defaultValue)
{
  for (int i = 0; i < a.length; i++)
    a[i] = defaultValue;
}
```

Type Erasure

- You cannot construct an array of a generic type:

```
public class Stack<E>
{
   private E[] elements;
   . . .
   public Stack()
   {
      elements = new E[MAX_SIZE]; // Error
   }
}
```

- Because the array construction expression `new E[]` would be erased to `new Object[]`.
- One remedy is to use an array list instead:

```
public class Stack<E>
{
   private ArrayList<E> elements;
   . . .
   public Stack()
   {
      elements = new ArrayList<E>(); // Ok
   }
   . . .
}
```

Type Erasure

- Use an array of objects and cast when reading elements from the array:

```
class Array<E>
{   private final Object[] arr;
    public final int length;
    // constructor
    public Array(int length)
    {    // Creates a new object array of the specified length
        arr = new Object[length];
        this.length = length;
    }

    // Method to get object present at index `i` in the array
    E get(int i) {
        @SuppressWarnings("unchecked")
        final E e = (E)arr[i];
        return e;
    }

    // Method to set a value `e` at index `i` in the array
    void set(int i, E e) {
        arr[i] = e;
    }

    @Override
    public String toString() {
        return Arrays.toString(arr);
    }
}
```

Type Erasure

- To use the Array class, we can do the following

```
class Main
{
   // Program to create a generic array in Java
   public static void main(String[] args)
   {
      final int length = 5;

      // create an Integer array of the given length
      Array<Integer> intArray = new Array(length);

      for (int i = 0; i < length; i++) {
         intArray.set(i, i + 1);
      }

      System.out.println(intArray);

      // create a string array of the given length
      Array<String> strArray = new Array(length);

      for (int i = 0; i < length; i++) {
         strArray.set(i, String.valueOf((char)(i + 65)));
      }

      System.out.println(strArray);
   }
}
```

Type Erasure

- Note that the component type of the array should be the *erasure* of the type parameter:

```
public class GenSet<E extends Foo> {
 // E has an upper bound of Foo
  private Foo[] a; // E erases to Foo, so use Foo[]

  public GenSet(int s) {
    a = new Foo[s];
  }
  ...
}
```

Type Erasure

- We can use the Reflection Array class to create an array of a generic type known only at runtime.
  Here, we're explicitly passing the Type information to the class constructor, which is further being passed to
  the Array.newInstance() reflection method.

```
// constructor
   public Array(Class<E> type, int length)
   {
      // Creates a new array with the specified type and length at runtime
      this.arr = (E[]) Array.newInstance(type, length);
      this.length = length;
   }
```

Type Erasure

- To use the this implementation of the Array class, we can do the following

```
class Main
{
   // Program to create a generic array in Java
   public static void main(String[] args)
   {
      final int length = 5;
      // create an Integer array of the given length
      Array<Integer> intArray = new Array(Integer.class, length);

      for (int i = 0; i < length; i++) {
         intArray.set(i, i + 1);
      }

      System.out.println(intArray);
      // create a string array of the given length
      Array<String> strArray = new Array(String.class, length);

      for (int i = 0; i < length; i++) {
         strArray.set(i, String.valueOf((char)(i + 65)));
      }
      System.out.println(strArray);
   }
}
```

Self Check 18.24

Could the `Stack` example be implemented as follows?

```
public class Stack<E>
{
  private E[] elements;
  . . .
  public Stack()
  {
    elements = (E[]) new Object[MAX_SIZE];
  }
  . . .
}
```

Answer: This code compiles (with a warning), but it is a poor technique. In the future, if type erasure no longer happens, the code will be *wrong*. The cast from `Object[]` to `String[]` will cause a class cast exception.

Self Check 18.25

The `ArrayList<E>` class has a method:

```
Object[] toArray()
```

Why doesn't the method return an `E[]`?

Answer: Internally, `ArrayList` uses an `Object[]` array. Because of type erasure, it can't make an `E[]` array. The best it can do is make a copy of its internal `Object[]` array.

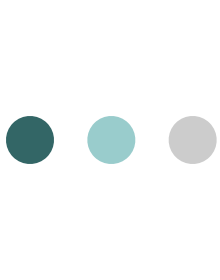
Self Check 18.26

The `ArrayList<E>` class has a second method:

```
E[] toArray(E[] a)
```

Why can this method return an array of type `E[]`?

Answer: It can use reflection to discover the element type of the parameter `a`, and then construct another array with that element type (or just call the `Arrays.copyOf` method).



CS 260

- Event-driven Programming and Graphical User Interfaces (GUIs) with Swing/AWT

- Reference:
  materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia

Why learn GUIs?

- Learn about event-driven programming techniques
- Practice learning and using a large, complex API
- A chance to see how it is designed and learn from it:
  - model-view separation
  - design patterns
  - refactoring vs. reimplementing an ailing API
- Because GUIs are neat!

- Caution: There is way more here than you can memorize.
  - Part of learning a large API is "letting go."
  - You won't memorize it all; you will look things up as you need them.
  - But you can learn the fundamental concepts and general ideas.

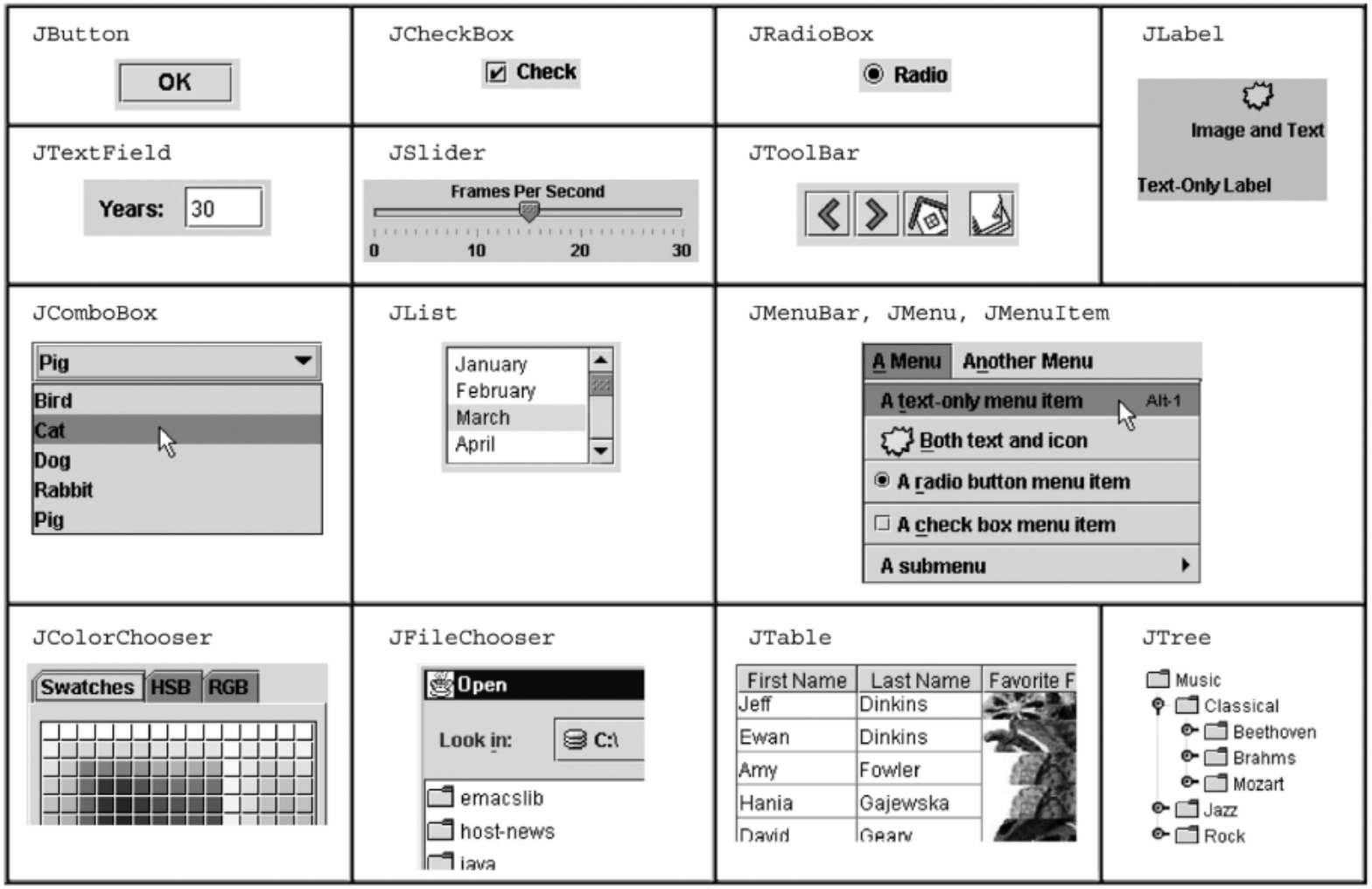
Java GUI History

- Abstract Windowing Toolkit (AWT): Sun's initial effort to create a set of cross-platform GUI classes. (JDK 1.0 - 1.1)
  - Maps general Java code to each operating system's real GUI system.
- Swing: A newer GUI library written from the ground up that allows much more powerful graphics and GUI construction. (JDK 1.2+)
  - Paints GUI controls itself pixel-by-pixel rather than handing off to OS.
  - Benefits: Features; compatibility; OO design.
  - Problem: Both exist in Java now; easy to get them mixed up; still have to use both in various places.

GUI terminology

- window: A first-class citizen of the graphical desktop.
  - Also called a top-level container.
  - examples: frame, dialog box, applet
- component: A GUI widget that resides in a window.
  - Also called controls in many other languages.
  - examples: button, text box, label
- container: A logical grouping for storing components.
  - examples: panel, box

Components


JButton
OK
JCheckBox
Check
JRadioBox
Radio
JLabel
Image and Text
Text-Only Label
JTextField
Years:
30
JSlider
Frames Per Second
0
10
20
30
JToolBar
JComboBox
Pig
Bird
Cat
Dog
Rabbit
Pig
JList
January
February
March
April
JMenuBar, JMenu, JMenuItem
A Menu
Another Menu
A text-only menu item
Alt-1
Both text and icon
A radio button menu item
A check box menu item
A submenu
JColorChooser
Swatches
HSB
RGB
JFileChooser
Open
Look in:
C:\
emacslib
host-news
java
JTable
First Name
Last Name
Favorite F
Jeff
Dinkins
Ewan
Dinkins
Amy
Fowler
Hania
Gajewska
David
Geary
JTree
Music
Classical
Beethoven
Brahms
Mozart
Jazz
Rock

Swing inheritance hierarchy

- Component (AWT)
  - Window
    - Frame
      - JFrame (Swing)
      - JDialog
  - Container
    - JComponent (Swing)
      - JButton
      - JColorChooser
      - JFileChooser
      - JComboBox
      - JLabel
      - JList
      - JMenuBar
      - JOptionPane
      - JPanel
      - JPopupMenu
      - JProgressBar
      - JScrollbar
      - JScrollPane
      - JSlider
      - JSpinner
      - JSplitPane
      - JTabbedPane
      - JTable
      - JToolbar
      - JTree
      - JTextArea
      - JTextField
      - …

```
import java.awt.*;
import javax.swing.*;
```

Component properties

- Each has a get (or is) accessor and a set modifier method.
- examples: getColor, setFont, setEnabled, isVisible

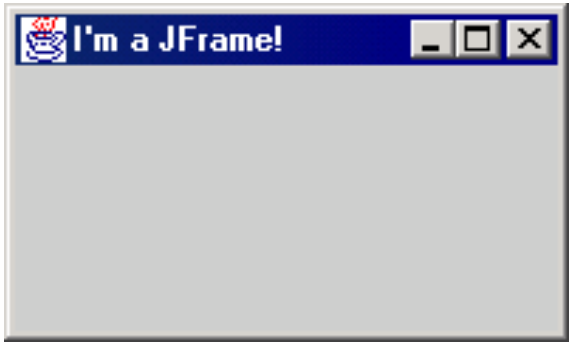
name	type	description
background	Color	background color behind component
border	Border	border line around component
enabled	boolean	whether it can be interacted with
focusable	boolean	whether key text can be typed on it
font	Font	font used for text in component
foreground	Color	foreground color of component
height, width	int	component's current size in pixels
visible	boolean	whether component can be seen
tooltip text	String	text shown when hovering mouse
size, minimum / maximum / preferred size	Dimension	various sizes, size limits, or desired sizes that the component may take

JFrame

- a graphical window to hold other components

- public JFrame()
  public JFrame(String title)
  Creates a frame with an optional title.
  - Call setVisible(true) to make a frame appear on the screen after creating it.

- public void add(Component comp)
  - Places the given component or container inside the frame.



More JFrame

- public void setDefaultCloseOperation(int op)
  - Makes the frame perform the given action when it closes.
  - Common value passed: JFrame.EXIT_ON_CLOSE
  - If not set, the program will never exit even if the frame is closed.
- public void setSize(int width, int height)
  Gives the frame a fixed size in pixels.
- public void pack()
  Resizes the frame to fit the components inside it snugly.