

Scientific Programming Using Object-Oriented Languages

Module 2: Objects and Classes

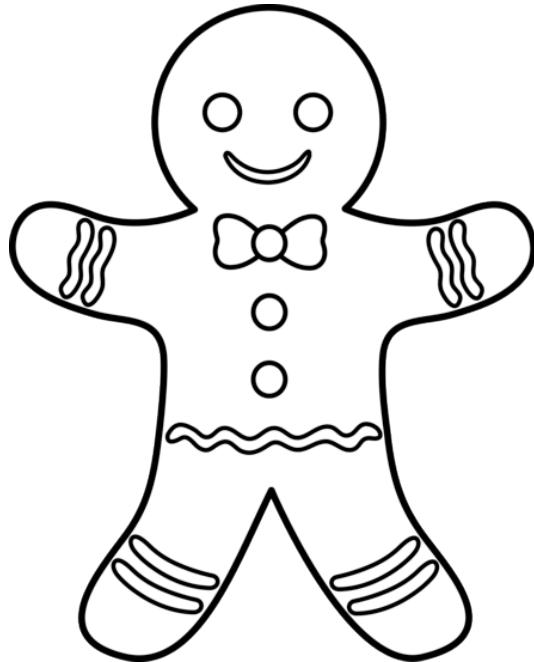
Aims of Module 2:

- Be able to define classes and use objects.
- Understand and apply the concept of encapsulation.
- Know when and how to use **static** methods and variables.
- Know when and how to use **final** variables.
- Know about and be able to use the `String` class and the numeric data-type wrapper classes.
- Know how to find out about other classes in the Java API (docs.oracle.com/javase/7/docs/api/).

Why Are Objects Used?

- Represent more complicated data-types than single numbers
- Represent objects from the real world
- Modularise code:
 - easier to understand
 - easier to extend
- Encapsulation:
 - objects can only be manipulated through controlled methods
 - protects user from unintended consequences of directly changing variables
- Polymorphism/inheritance
 - but that's another module!

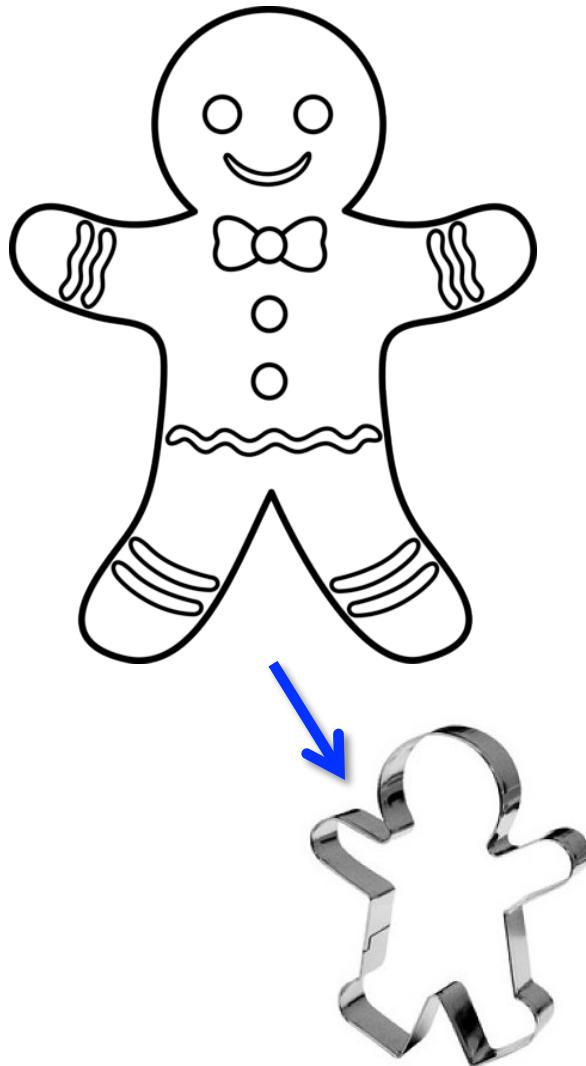
Classes & Objects: The Class



A *class* is a data type that can represent more complicated entities than simply single real or integer numbers, characters or boolean variables.

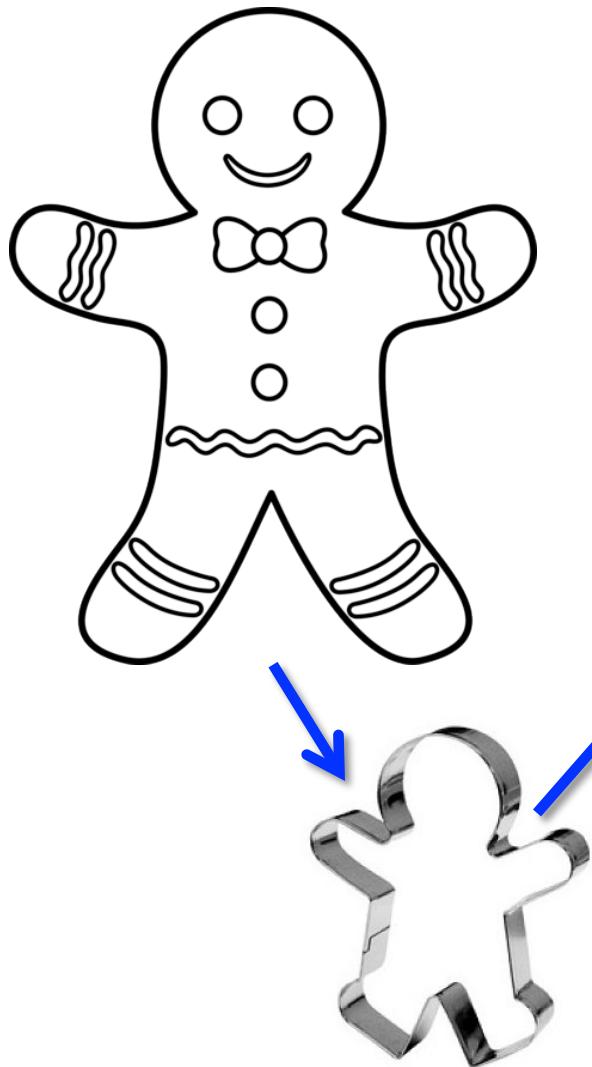
You create objects using the “blueprint” defined by the class.

Classes & Objects: The Constructor



A *constructor* is used to set up an object when we tell Java to create a new object using the `new` command. The constructor is defined within the class and populates the structure of the object when you first *instantiate* it.

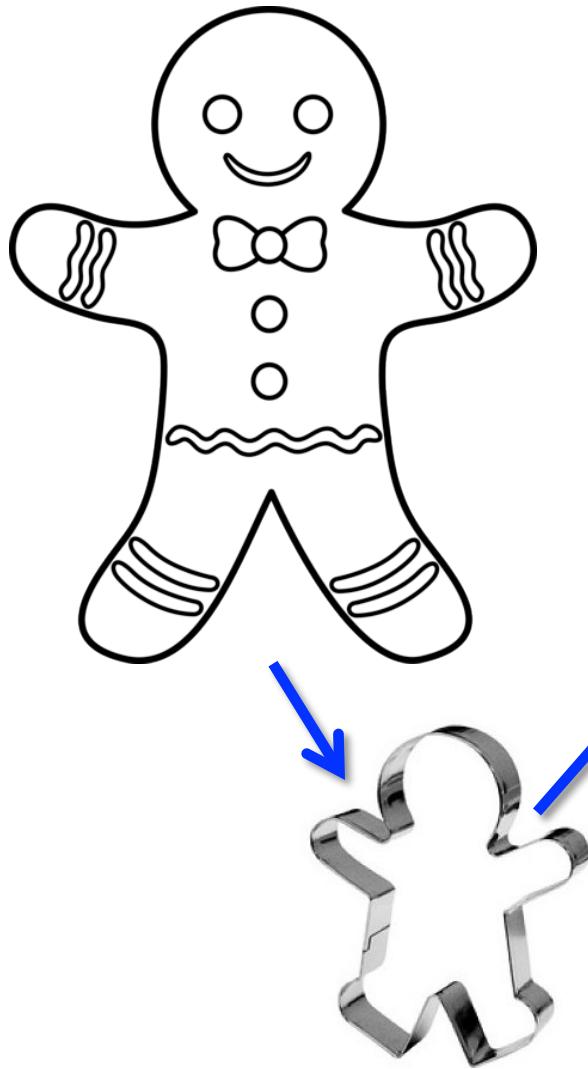
Classes & Objects: Instantiation



Creating an object using a constructor from a given class is called *instantiation*. This creates a new *instance* of that class in the form of the new object.



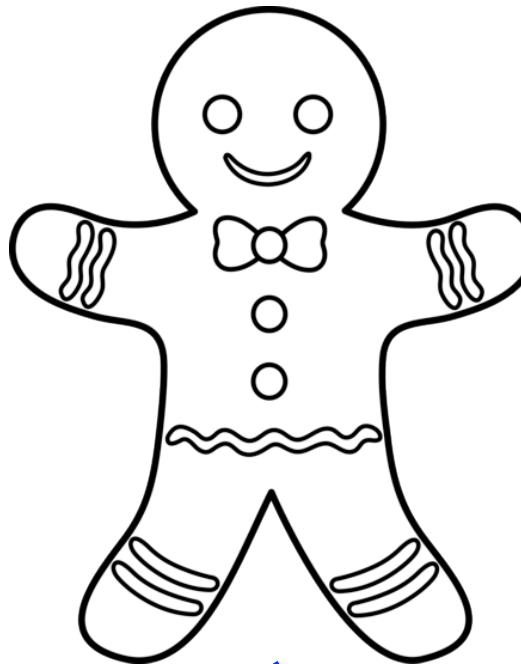
Classes & Objects: A New Object



Once you instantiate a new object using the constructor, you now have an object whose “layout” follows the “blueprint” defined by the class. The new object contains all the *methods* defined by that class as well as the default member variables.



Classes & Objects: Methods

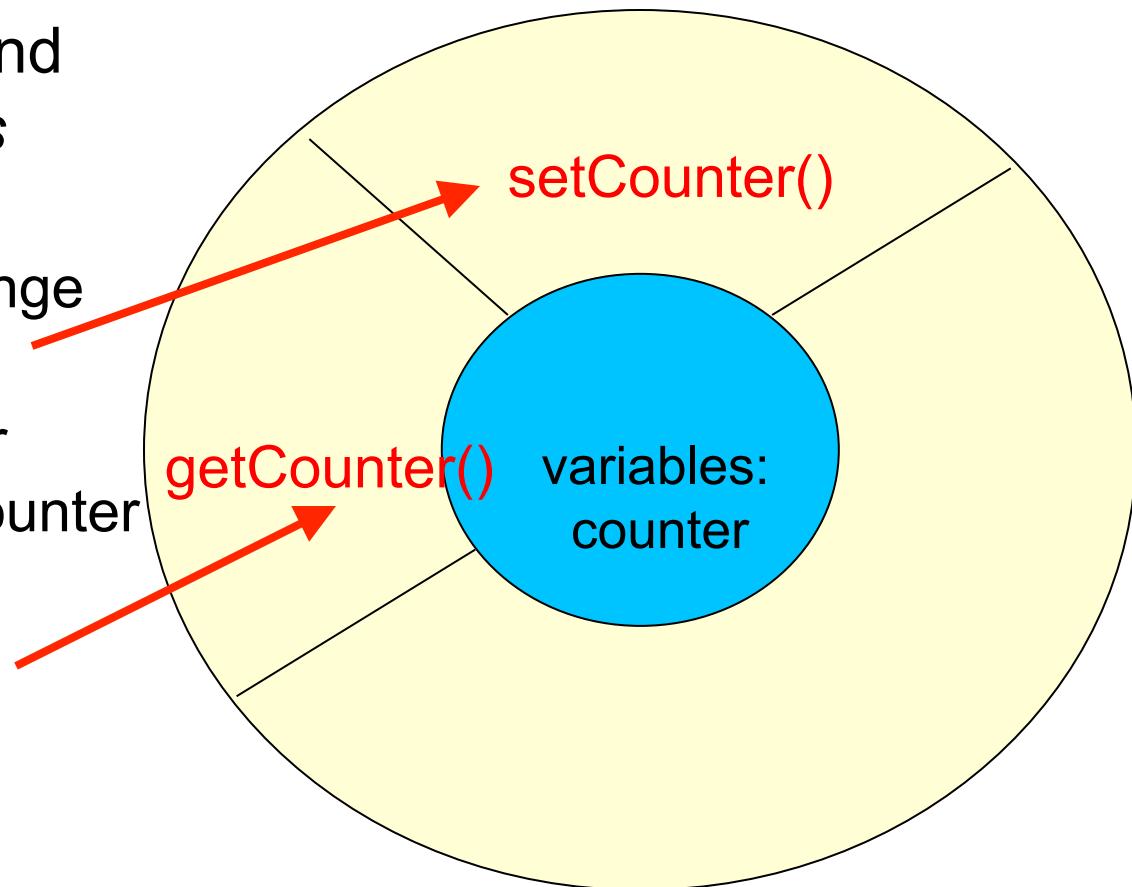


You can now manipulate the data contained within the object using the *methods* associated with that class. These “functions” allow you to modify the state of the object – such as passing data back and forth – or make specific calculations defined by the class.



Encapsulation

- Communicate with an object by sending and receiving *messages*
 - send message to setCounter to change counter to 10
 - ask getCounter for current value of counter



A Simple Class

```
public class SimpleCounter {  
  
    int counter;          // member variable  
    public SimpleCounter() {} // constructor (ignore for now)  
  
    int getCounter() {return counter;} // method to set value  
    void setCounter(int val) {counter = val;} // retrieve value  
}
```

Using Our Class

```
public class TestSimpleCounter {  
  
    public static void main(String[] args) {  
  
        SimpleCounter count1 = new SimpleCounter();  
        count1.setCounter(10);  
        System.out.println("value of counter is  
                           "+count1.getCounter());  
  
    }  
  
}
```

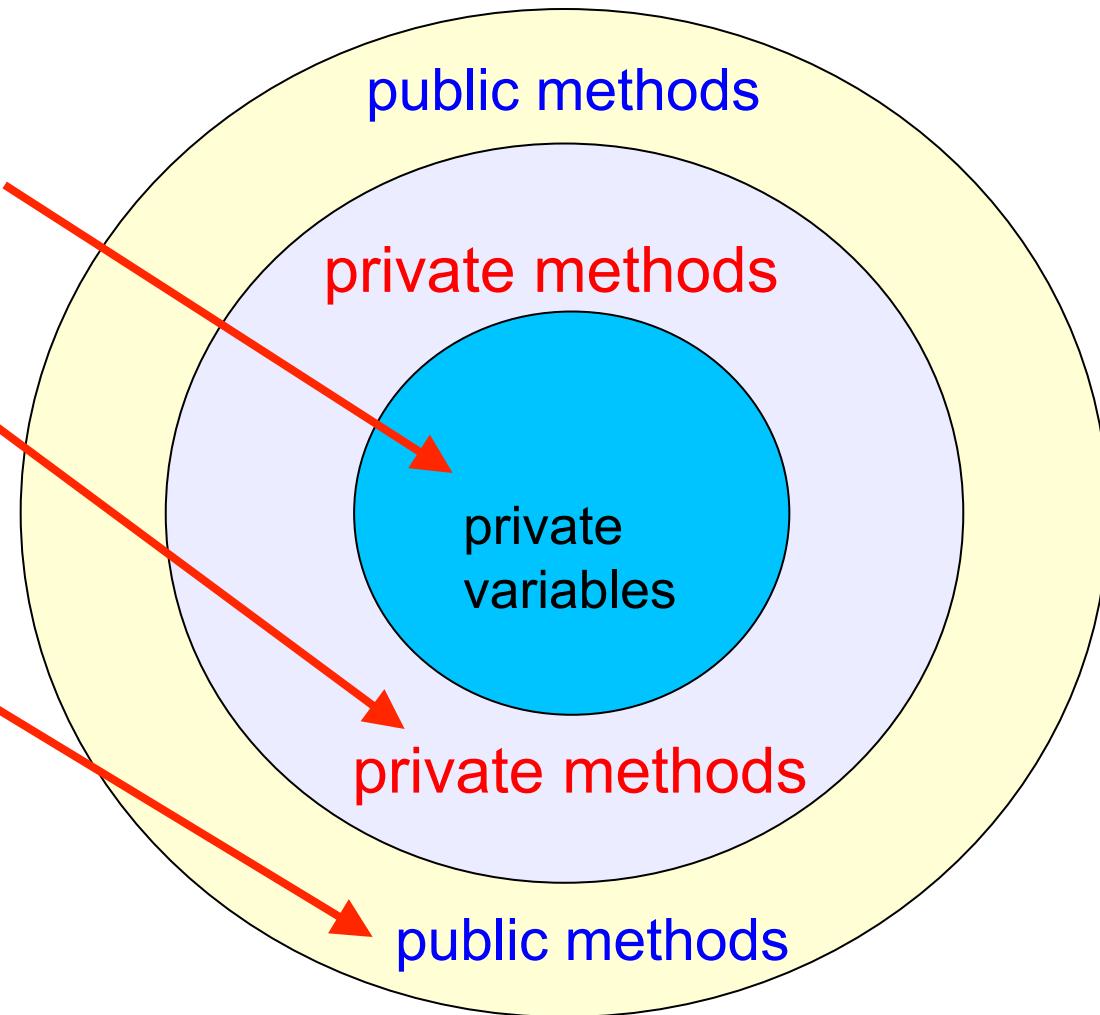
Constructors

```
public SimpleCounter() { }
public SimpleCounter(int val) { counter = val; }

SimpleCounter count1 = new SimpleCounter(); // set to 0
SimpleCounter count1 = new SimpleCounter(3); // set to 3
```

Encapsulation

- Prevent direct access to variables by making them private
- Can also have private methods, only usable from within the class
- Other classes/objects can only access this one through its public methods



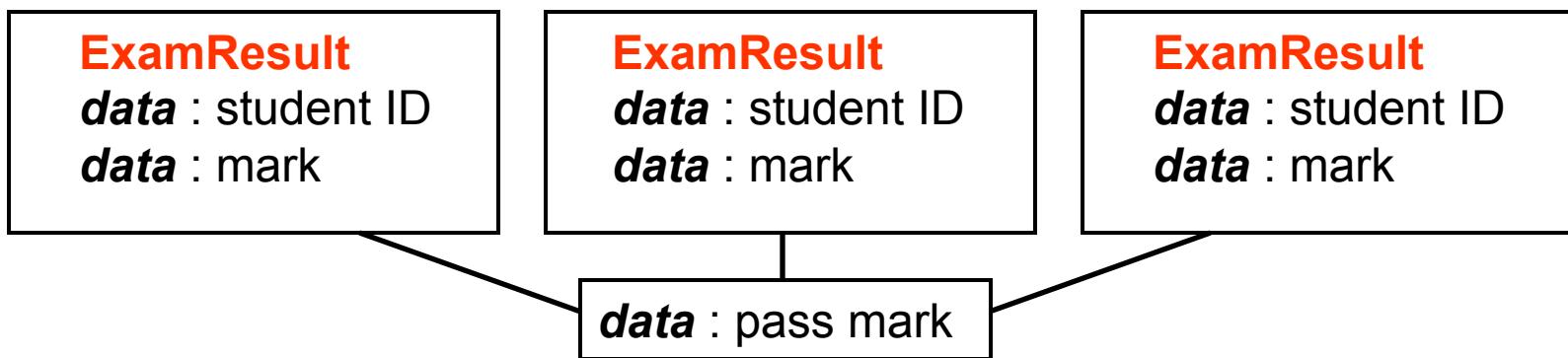
Access Control

- Why not `count1.counter = 10` ?
- How to stop people from doing this?

```
public class SimpleCounter {  
    private int counter;  
    public SimpleCounter() {}  
  
    int getCounter() { return counter; }  
  
    void setCounter(int val) { counter = val; }  
}
```

Static Variables

- In general, each instance of an object of a particular type holds its own set of data that can differ from one instance to another.
- But there are occasions when you want the universe to contain a single data instance which is “**shared**” or “**viewed**” by all object instances.
- For example in a piece of teaching software we might have :



Static Variables

- This is achieved by declaring such variables to be static
- Another example :

```
public class SimpleCounter {  
    private static int max = 1000; // same val for every SimpleCounter  
    private int counter;  
    public SimpleCounter() {}  
  
    int getCounter() {return counter;}  
  
    void setCounter(int val) {  
        if (val<=max) counter = val; // only change if new value < max  
    }  
}
```

can be referred to using “class_name.variable_name”,
e.g. “SimpleCounter.max”

Static Methods

- **static** methods do not depend on any non-static variables.
- In other words, the method does not depend on the unique state of a particular object.
- A static method can be used without even bothering to create an object at all by invoking
“`class_name.method_name(args)`”
- For example :

```
double cos_90 = Math.cos(Math.toRadians(90));
```

The results of these math functions depend only on the arguments passed and not on the state of the object

“this”

- The keyword “**this**” is used to refer to the current object instance.

```
public static SimpleCounter add(SimpleCounter x, SimpleCounter y) {  
    int sum = x.counter + y.counter;  
    return new SimpleCounter(sum);  
}
```

```
public SimpleCounter add(SimpleCounter x) {  
    int sum = x.counter + this.counter;  
    return new SimpleCounter(sum);  
}
```

```
SimpleCounter sumA = sc1.add(sc2) // non-static  
SimpleCounter sumB = SimpleCounter.add(sc1, sc2)
```

Rewrite to avoid code duplication.
HINT !!! Exercises ...

```
public SimpleCounter add(SimpleCounter x) {  
    return new add(this, x);  call static version  
}
```

Final Variables

- Variables can be declared final to prevent them being modified (either accidentally or misguidedly) during program execution.
- For example :

```
public static final double PI;
```

which can be accessed by:

should never change ...

```
Math.PI;
```

- Physics simulations contain lots of constants : fundamental and specific.

Converting Objects to Strings

- By defining a `toString` method with the following signature, it becomes particularly simple to print the state of an object to the screen. For example, for our `SimpleCounter` class :

```
public String toString() {  
    return "counter = "+counter+", max = "+max;  
}
```

SimpleCounter class definition

This method is invoked
when the object is
converted to a string:

Take care to provide an
informative yet concise string
version of your object.

```
SimpleCounter c = new SimpleCounter();  
System.out.println("state of c:"+c);
```

Calling Code

- See Module 2 notes for more details of the `String` class.

Java API: docs.oracle.com/javase/7/docs/api/

Overview (Java Platform SE) 

docs.oracle.com/javase/7/docs/api/

Apps Google Maps YouTube News Imported From Firefox

Java™ Platform Standard Ed. 7

All Classes Packages

- java.awt
- java.awt.color
- java.awt.datatransfer
- java.awt.dnd
- java.awt.event
- java.awt.font
- java.awt.geom
- java.awt.im
- java.awt.im.spi
- java.awt.image

All Classes

- AbstractAction
- AbstractAnnotationValueVisitor6
- AbstractAnnotationValueVisitor7
- AbstractBorder
- AbstractButton
- AbstractCellEditor
- AbstractCollection
- AbstractColorChooserPanel
- AbstractDocument
- AbstractDocument.AttributeContext
- AbstractDocument.Content
- AbstractDocument.ElementEdit
- AbstractElementVisitor6
- AbstractElementVisitor7
- AbstractExecutorService
- AbstractInterruptibleChannel
- AbstractLayoutCache
- AbstractLayoutCache.NodeDimensions
- AbstractList
- AbstractListModel
- AbstractMap
- AbstractMap.SimpleEntry
- AbstractMap.SimpleImmutableEntry
- AbstractMarshallerImpl
- AbstractMethodError
- AbstractOwnableSynchronizer
- AbstractPreferences
- AbstractProcessor
- AbstractQueue
- AbstractQueuedLongSynchronizer
- AbstractQueuedSynchronizer
- AbstractRegionPainter
- AbstractRegionPainter.PaintContext
- AbstractRegionPainter.PaintContext.CacheMode
- AbstractScriptEngine
- AbstractSelectableChannel
- AbstractSelectionKey
- AbstractSelector
- AbstractSequentialList
- AbstractSet
- AbstractSpinnerModel

Overview Package Class Use Tree Deprecated Index Help

Prev Next Frames No Frames

Java™ Platform, Standard Edition 7 API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

See: Description

Packages

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing beans -- components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.annotation	Provides library support for the Java programming language annotation facility.
java.lang.instrument	Provides services that allow Java programming language agents to instrument programs running on the JVM.
java.lang.invoke	The <code>java.lang.invoke</code> package contains dynamic language support provided directly by the Java core class libraries and virtual machine.
java.lang.management	Provides the management interfaces for monitoring and management of the Java virtual machine and other components in the Java runtime.
java.lang.ref	Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
java.lang.reflect	Provides classes and interfaces for obtaining reflective information about classes and objects.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (<code>BigInteger</code>) and arbitrary-precision decimal arithmetic (<code>BigDecimal</code>).
java.net	Provides the classes for implementing networking applications.
java.nio	Defines buffers, which are containers for data, and provides an overview of the other NIO packages.