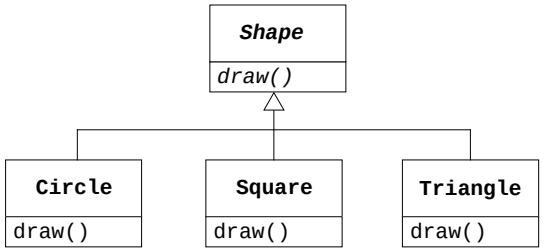
Run-Time Type Identification

Intent

Lets you find the exact type of an object even if your handle is a base type of your object.

Polymorphism

Consider the familiar example:



A code excerpt might be:

```
public abstract class Shape {
   public abstract void draw();
}

public class Circle extends Shape {
   public void draw() {
      // Would really draw into a Graphics object..
      System.out.println("Circle.draw()");
   }
}

// Classes Square, Triangle: Accordingly.
```

A client might create a collection of Shape objects:

```
// In the body of a method:
ArrayList shapes = new ArrayList();
shapes.add(new Circle());
shapes.add(new Square());
shapes.add(new Triangle());
```


A client uses the above collection of Shape objects:

```
// In the body of a method:
ArrayList shapes = ...; // Given as argument,
                        //or created locally.

for (Iterator i = shapes.iterator(); i.hasNext(); ) {
   Object o = i.next();// declared return type of
                       // method next(): Object
   Shape s = (Shape) o;// downcast
   s.draw();
}
```

The actual type contained in variable s determines which instance of method draw() is executed when applying it to s.

If the collection contains an object that hasn't Shape as its base class then the unchecked ClassCastExcpetion is thrown.

But what if you want to highlight all Triangle objects in a given collection of a Shape objects?

Downcast

Given a Shape object, you can downcast it to a Triangle object. If it is not a Triangle object, a ClassCastException is thrown:

```
Shape s = ...;
Triangle t;
try {
   t = (Triangle) s;
} catch (ClassCastException e) { ... }
```

In this approach, you try the downcast, and if it does not fail, then you know that a given object's type is the one of a particular subtype. This is not always a good style, however.

Checking for an Instance of a Particular Type

Given a collection of `Shape` objects, you can check if an item of the collection is a `Triangle` object by using the `instanceof` operator:

```
// in a method:
ArrayList shapes = ...; // provide a collection of shapes

for (Iterator i = shapes.iterator(); i.hasNext(); ) {
   Shape s = (Shape) i.next();
   if (s instanceof Triangle) {
      // change color
   }
}
```


However, suppose you want to count the concrete subtypes of a collection of shapes. Using the `instanceof` operator, you end up in a client class with something like:

```
if (s instanceof Circle)   { /* increment circle count   */}
if (s instanceof Square)   { /* increment square count   */}
if (s instanceof Triangle) { /* increment trianlge count */}
```

If you later add a new subtype of `Shape`, you'll have to modify the client code from above:

```
if (s instanceof Circle)   { /* increment circle count   */}
if (s instanceof Square)   { /* increment square count   */}
if (s instanceof Triangle) { /* increment trianlge count */}
if (s instanceof Line)     { /* increment line count     */}
```


The Class Object

Every object in your Java program is an instance of a class. Upon loading the code for a given class, the class itself is represented within the Java program by its *class object*. The class object is an instance of class `Class`.

The class object is created if:

- an instance of that class is created for the first time, or
- you create the class object explicitly.

Explicit Creation/Access of the Class Object. You can perform explicit class object creation or access to via:

- The static `Class.forName()` method:
  `Class cl = Class.forName("`*`fully_quallyfied_Classname`*`");`
- The *class literal*:
  `Class cl =` *`Classname`*`.class;`

Class literals are checked by the compiler at compile-time. They are safer, though, but less flexible. `Class.forName()` is more flexible. For instance, you can provide the string naming the class during execution of the program. `Class.forName()` may throw a checked `ClassNotFoundException`.

A Dynamic Instanceof Operator: Class.isInstance()

Method `Class.isInstance(Object)` allows to check if a given object belongs to a given class or one of its descendants. Counting the `Shape` objects can then be done as follows:

```
class Counter {
   int i;
}
```

In the body of some method:

```
Class[] shapeTypes =
   { Shape.class, Circle.class, Square.class, Triangle.class }
Map map = new HashMap();
for (int i=0; i < shapeTypes.length; i++) {
   map.put(shapeTypes[i], new Counter());
}
```

Array `shapeTypes` contains the set of class objects for the concrete subclasses of class `Shape`, as well as the class object for class `Shape`.

A `HashMap` object is initialized with class objects as the keys, and `Counter` objects as their values.

Given a set of shapes, lets count the total of `Shape` objects as well as the total of each subclass of `Shape`:

```
// in the body of some method:
ArrayList shapes = ...;

for (Iterator i = shapes.iterator(); i.hasNext(); ) {
   Object o = i.next();
   for (int j=0; j < shapeTypes.length; j++) {
      if (shapeTypes[j].isInstance(o)) {
         ((Counter) map.get(shapeTypes[j])).i++;
      }
   }
}
```

Notice that the outer for-loop need not be changed in the case of modifying the array `shapeTypes` of class object.

Notice also that, given for example an instance of a `Circle`, the counters for `Shape` objects and for `Circle` objects are incremented.

Method	Description
`Field[] getFields()`	Returns an array containing `Field` objects reflecting all the accessible public fields of the class or interface represented by this `Class` object.
`Method[] getMethods()`	Returns an array containing `Method` objects reflecting all the public member methods of the class or interface represented by this `Class` object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.
`Class[] getClasses()`	Returns an array containing `Class` objects representing all the public classes and interfaces that are *members* of the class represented by this `Class` object. This includes public class and interface members inherited from superclasses and public class and interface members declared by the class.

Reflection: Run-Time Class Information

Given an instance of class `Class`, you can get the complete type information for objects represented by that instance of class `Class`. for that purpose, some of methods of class `Class` are:

Classes `Constructor`, `Field`, and `Method` belong to package `java.lang.reflect`.

Method	Description
`Class[] getInterfaces()`	Determines the interfaces *implemented* by the class or interface represented by this object.
`Class getSuperclass()`	Returns the `Class` representing the superclass of the entity (class, interface, primitive type or `void`) represented by this `Class`. If this object represents either the `Object` class, an interface, a primitive type, or `void`, then `null` is returned. If this object represents an array class then the `Class` object representing the `Object` class is returned.
`Constructor[] getConstructors()`	Returns an array containing `Constructor` objects reflecting all the public constructors of the class represented by this `Class` object.