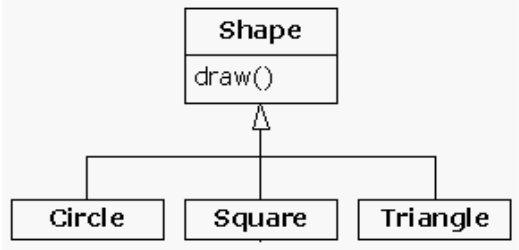
Run-Time Type Identification and Reflection

Advanced Programming

Which object do I draw?



Which object do I draw?

```
public class Shapes {
    public static void main(String[] args) {
        ArrayList s = new ArrayList();
        s.add(new Circle());
        s.add(new Square());
        s.add(new Triangle());
        Iterator e = s.iterator();
        while(e.hasNext()) (Shape)e.next()).draw();
  }
}
```


Motivation

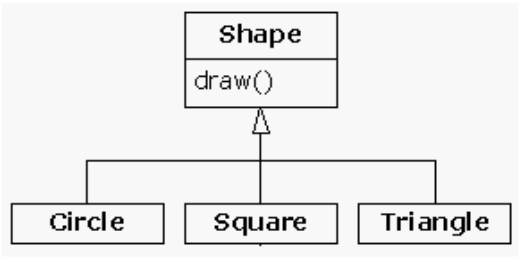
- Since all Java classes are derived from Object, it's easy to **mix objects of different types** together into a collection
- For Example: retrieving objects from collections creates problem, because the **actual type of the objects is lost**
- Solution: RTTI is used to reveal the **true types at run time**

RTTI

- RTTI lets you find the **exact type of an object** when you only have a **reference to the base type**

RTTI with polymorphism

- Consider the class hierarchy that uses polymorphism.
- The generic type is the base class Shape, and the specific derived types are Circle, Square, and Triangle:



Goal of RTTI

- You can **manipulate references** to the **base type** (Shape, in this case) and
- when extending the program by adding a new class (Triangle, derived from Shape, for example), the **original code where the reference is used does not change**

RTTI examples

- Casting
- instanceof operator

Casting

```
import java.util.*;
class Shape {
    void draw() {
        System.out.println(this + ".draw()");
    }
}
class Circle extends Shape {
     public String toString() { return
"Circle"; }
}
class Square extends Shape {
     public String toString() { return
"Square"; }
}

class Triangle extends Shape {d
     public String toString() { return
"Triangle"; }
}
```

```
public class Shapes {

    public static void main(String[] args) {

        ArrayList s = new ArrayList();

        s.add(new Circle());

        s.add(new Square());

        s.add(new Triangle());

       Iterator e = s.iterator();

        while(e.hasNext()) (Shape)e.next()).draw();

  }
}
```

I just need to cast for the base, then polymorphism applies, e.g.,

Shape x = new Circle()

e.next() returns an object of type Object

When I add a new child the original code does not change

Interpreting the example

- In

((Shape)e.next()).draw();

- we mean the next object is referenced by a reference variable of type Shape:

Shape aShape

- Then we pass an object of the children type at run time (the entry in the array)

Shape aShape = new Circle();

- We have manipulated the base with casting and we have used polymorphism with the instantiation above

Interpreting the example

- In main( ), specific types of Shape are created and then added to an ArrayList. This is the point at which the **downcast** occurs because the **ArrayList holds only Object references**
  - when we access the ArrayList we just get Object: we have lost the specialisation of the objects in the array
- Therefore next( ) naturally produces an Object reference (Object x)
- So a **cast to Shape** is necessary
- Down casting is the first example of RTTI, since in Java all casts are **checked** for correctness **at run-time**!

Note on casting types

- (**Downcasting**) A conversion from type Object to type Shape requires a **run-time check** to make sure that the run-time value is actually an instance of class Shape or one of its subclasses; if it is not, an **exception is thrown (ClassCastException)**.

  Shape aShape= (Shape) getDog();

  - if the returning object of getDog() is not a subtype of Shape, **this throws an exception**
- (**Upcasting**) A conversion from type Shape to type Object requires **no run-time action**; Shape is a subclass of Object, so any reference produced by an expression of type Shape is a valid reference value of type Object. In this case, the **reference variable points to the internal object** of the object of Shape of type Object

RTTI with

- instanceof operator

Instanceof

- The operator instanceof **compares an object to a specified type and returns true or false**
  - In **inheritance**, as derived classes (Circle and Triangle) are a base class (Shape), it can be applied **without casting**
  - It returns false when given a **null** value or the object is **not of a given class**

Example: instanceof()

```
class InstanceofTest {
   public static void main(String[] args) {

   Shape obj1 = new Shape();
   Shape obj2 = new Circle();

   System.out.println("obj1 instanceof Shape: " + (obj1 instanceof Shape));
   System.out.println("obj1 instanceof Circle: " + (obj1 instanceof Circle));
   System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
   System.out.println("obj2 instanceof Shape: " + (obj2 instanceof Shape));
   System.out.println("obj2 instanceof Circle: " + (obj2 instanceof Circle));
   System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
   }
}
class Shape{}
class Circle extends Shape implements MyInterface{}
interface MyInterface{}
```


instanceof()

Output:

obj1 instanceof Shape: true

obj1 instanceof Circle: false

obj1 instanceof MyInterface: false

obj2 instanceof Shape: true

obj2 instanceof Circle: true

obj2 instanceof MyInterface: true

Circle is a child of Shape so if I ask whether it is of type Shape I get true

Do not use instanceof instead of polymorphism!

```
class Shape {
    void draw() {
        System.out.println(“Shape”);
    }
}
class Circle extends Shape {
     public String drawCircle() {System.out.println("Circle”); }
}
class Square extends Shape {
     public String drawSquare() {System.out.println(“Square”); }
}
public static void draw(Shape o){
    if (o instanceof Circle){
        Circle circle = (Circle) o;
        o.drawCircle();}
    else if (o instanceof Square){
        Square circle = (Square) o;
        o.drawSquare();}
}
```


RTTI vs reflection API

- With **RTTI we need to have all the types we use at run time available at compile time**
- Many times this is not possible:
  - If we receive data that represents classes remotely, at compile time we do not have them
    - e.g. with Remote Method Invocation (RMI) we can call methods distributed on remote machines. Locally at compile time, we do not know the types used by these methods

Reflection

- With Reflection we do not need it
- We will see some examples:
  - getClass() from Object
  - The class Class
  - The reflection API

Reflection API

- The reflection API comprises the
- **java.lang.reflect package** and
- **java.lang.Class**

java.lang.Class

http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html

- Instances of the class Class represent classes and interfaces of an application at run time
  - There is an object of type Class for each class or interface whose objects exist at run time
  - Every **array** is reflected as a Class object that is shared by all arrays with the **same element type and number of dimensions**
  - The **primitive Java types** (boolean, byte, char, short, int, long, float, and double), and the keyword **void** are also represented as Class objects

java.lang.Class

- For every type of object, the Java virtual machine instantiates an i**mmutable instance of java.lang.Class**
- It provides methods to examine the runtime properties of an object including its members and type information
- Class also provides the ability to create new classes and objects

Understanding java.lang.Class

- Every time we write and compile a new class one object of Class is created
- It is saved in a file .class and called with the same name of the class
- At run time when we want to create a given object of the class, the JVM first checks if the object of **Class** has been already loaded
- If not, the Loader searches for the file .class with the same name of the class in the file system

Getting the objects of Class

- If an instance of an object is available, then the simplest way to get its class is to invoke **Object.getClass()**:

```
Class c = "foo".getClass();
```

- Returns the class String as object of Class

```
import java.util.HashSet;
import java.util.Set;
Set<String> s = new HashSet<String>();
Class c = s.getClass();
```

- java.util.Set is an interface to an object of type java.util.HashSet. The value returned by getClass() is the class **java.util.HashSet**
- The generic <Integer> is discarded (we will see the type erasure later on)

Example

```
public class Animal {
       private String name;
       public Animal(String name) { this.name = name; }
       public String getName() { return name; }
       public void setName(String name) { this.name = name; }
       public String toString() { return "<" + name + ">"; }
}
public class Cat extends Animal {
       public Cat(String name) {
        super(name);
       }
}
public class Dog extends Animal {
   public Dog(String name) {
          super(name);
        }
}
```


Example

```
import java.util.*;

public class BagOfObjects {
   private ArrayList listOfObjects = new ArrayList();
   private int noCats = 0;
   private int noDogs = 0;
   private int noOthers = 0;

   public void add(Object o) {
      if (o instanceof Cat)
         noCats++;
      else if (o instanceof Dog)
         noDogs++;
      else
         noOthers++;
      listOfObjects.add(o);
   }
```

Using the operator **instanceof** we detect the true type of the object being added to the *bag*. This is fine as we are using instanceof explicitly and not with method overriding

Example

```
// continued class BagOfObjects…
    public String toString() {
       String res = "In total there are: \n";
       res += " " + noCats + " cats,\n";
       res += " " + noDogs + " dogs, and\n";
       res += " " + noOthers + " others\n\n";
       for (int i = 0; i < listOfObjects.size(); i++) {
          Object t = listOfObjects.get(i);
          res += "\tElement " + i + " is an object of " + t.getClass();
          if (t instanceof Dog || t instanceof Cat) {
             Animal a = (Animal) t;
             res += "\t --- Name: " + a.getName() + " \n ";
          }
          else
             res += "\n";
       }
       return res;
    }
}
```

The method getClass(), **inherited from Object** indirectly accesses the class object of t in order to display its class name **at run time**

We check at run time the type of t, then we **downcast** it to Animal **in a safe way** as t **is a reference of type Object** . Then we apply polymorphism to get the right name of the children with **getName**() **at run time**.

Example

```
public class BagTest {

   public static void main(String argv[]) {
      BagOfObjects aBag = new BagOfObjects();

      aBag.add(new Object());
      aBag.add(new Cat("Nusa"));
      aBag.add(new Cat("Garfield"));
      aBag.add(new Dog("Bello"));

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

Interpreting the example

- The method **toString() of Object** is overridden by the **toString() of BagOfObjects** and
- the console displays the nested information returned with the String "res"

Getting the objects of Class

- The **static method forName(String s) of Class** receives a string with the exact name of a class and returns a reference to the **object of Class for that type**

```
Class.forName("Shape");
```

- returns a reference to the object of Class of type Shape

Getting the name of objects of Class

- getName() returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String

```
this.getClass().getName()
```


Getting the attributes of a class

- The class Field has methods to get attributes names, modifiers, and values at run-time

```
getModifiers()
this.getClass().getDeclaredFields()
get(Object o)
getName()
```

Example

```
public final class Sample {
   private String fName;
   public Sample(String fName){
      super();
      this.fName = fName;
   }
   public String toString() {
      StringBuffer result = new StringBuffer();
      String newLine = System.getProperty("line.separator");
      result.append( this.getClass().getName() );
      result.append( " Object {" );
      result.append(newLine);

      java.lang.reflect.Field[] fields = this.getClass().getDeclaredFields();

      for ( int fieldId=0; fieldId < fields.length; ++fieldId ) {
        result.append(" ");
        try {
            result.append( fields[fieldId].getName() );
            result.append(": ");
            result.append( fields[fieldId].get(this) );
```

get the String name

get the value of the field in the "this" object

Exercise cont.

```
      …    } catch ( IllegalAccessException ex ) { System.out.println(ex);}
            result.append(newLine);
       }
      result.append("}");
      return result.toString();
      }

      public static void main ( String[] arguments ) {
         Sample  sample = new Sample( "CIAO");
         System.out.println(sample);
      }
}
```

An IllegalAccessException is thrown when an application tries to reflectively create an instance (other than an array), set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method or constructor.

Write the output!

Output

Sample Object {

fName: CIAO

}