JAVA
INTERFACCE ED EREDITARIETÀ

Interfacce

- Esistono situazioni in cui gruppi di programmatori devono concordare su come il loro codice interagisce
  - Esempio: come la parte grafica di una applicazione interagisce con la parte operativa
- Serve un "contratto" che rappresenti queste specifiche
  - Le interfacce rappresentano tale contratto

Interfacce

- In Java, un'interfaccia è un **tipo**
  - E' simile ad una classe
  - Può contenere (principalmente) **solo** costanti e metodi non implementati
  - Non può contenere metodi implementati
  - Non possono essere istanziate

Interfacce
Dichiarazione

```
public interface Shape {
    // costanti

    // metodi senza implementazione
    int getPerimeter();
    int getArea();

}
```

Interfacce Implementazione

- Un'interfaccia viene utilizzata nel momento in cui viene scritta una classe che la implementa
- Si utilizza la keyword implements, di seguito alla dichiarazione della classe
- Quando una classe implementa un interfaccia, deve fornire un'implementazione per tutti i metodi dichiarati in quell'interfaccia

Interfacce Implementazione

```
public interface Shape {
    int getPerimeter();
    int getArea();

}
```

```
public class Rectangle {

    public int width;
    public int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }

}
```

Interfacce Implementazione

```java
public interface Shape {
    int getPerimeter();
    int getArea();

}
```

```java
public class Rectangle implements Shape {

    public int width;
    public int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getPerimeter() {
        return (width + height)*2;
    }

    public int getArea() {
        return width * height;
    }

}
```

Interfacce Implementazione

- Una classe può implementare più di un interfaccia
  - L'unico vincolo è che implementi tutti i metodi che eredita dalle interfacce
- Dopo implements si può specificare una lista di interfacce, separate da virgola

```
public class Rectangle implements Shape, Relatable {

    ...

}
```

Interfacce

- In Java, definire un interfaccia equivale a definire un tipo
- E' possibile utilizzare l'interfaccia in una dichiarazione

```
Shape aShape = new Rectangle(5, 3);
Relatable aRelatableObj = new Rectangle(5, 3);
```

- Quando si assegna il valore, questo deve essere un'istanza di una classe che implementa quell'interfaccia

Interfacce

- Anche se l'istanza è di un certo tipo, fa fede quello che è stato usato nella dichiarazione
  - In questo caso, un'interfaccia
- Saranno visibili solo i metodi dichiarati in quell'interfaccia

```
Shape aShape = new Rectangle(5, 3);
Relatable aRelatableObj = new Rectangle(5, 3);

aShape.getArea();
aShape.isBiggerThan( aRelatableObj );

aRelatableObj.getArea();
aRelatableObj.isBiggerThan( aShape );
```

Interfacce

- Grazie alle interfacce, è possibile scrivere codice che opera in modo agnostico rispetto all'implementazione
  - Questo permette di cambiare l'oggetto utilizzato, senza dover riscrivere codice

```
public int doSomething(Shape shape) {
     //...
}
```

```
Shape myShape = new Rectangle(5, 3);

doSomething(myShape);
```

```
Shape myShape = new Circle(4);

doSomething(myShape);
```

Interfacce
Default Methods

- I **Default Methods** permettono di aggiungere nuove funzionalità alle interfacce, senza "rompere" la compatibilità con le vecchie versioni
- Si tratta di metodi con implementazione, dichiarati nelle interfacce
  - Si dichiarano con la keyword **default**
  - Hanno implicitamente visibilità **public**

API

- Application Programming Interface
- Insieme di interfacce che rappresentano la funzionalità di un software o di una tecnologia
  - E.g. JPA è una API standard di Java, implementata da diverse librerie (Hibernate, EclipseLink, …)
- E' un modo per esporre una funzionalità senza fornire i dettagli di come è implementata

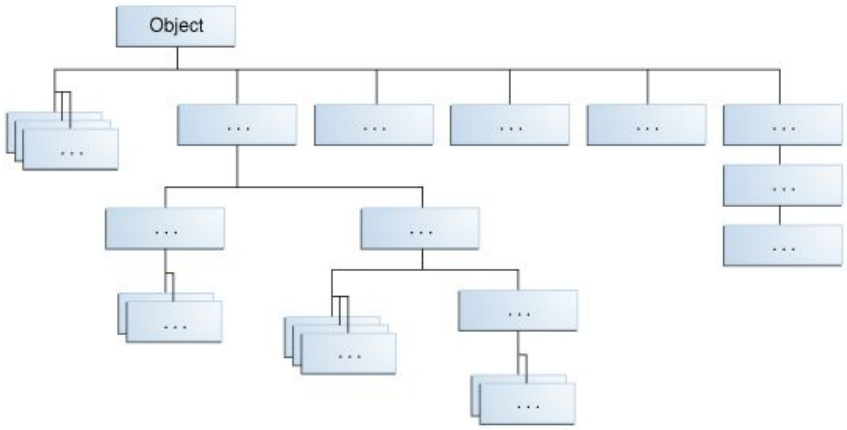
Ereditarietà

Ereditarietà

- Nella programmazione ad oggetti, una classe può ereditare da un'altra, utilizzando la keyword **extends**
  - Si stabilisce una relazione *padre-figlio* tra due classi
- La classe padre è chiamata **superclasse**, la classe figlio **sottoclasse**
  - Quando si eredita, la classe figlio ottiene l'accesso a campi e metodi della classe padre, **se la visibilità lo permette**

Ereditarietà
Gerarchia

- Ogni classe può ereditare direttamente solo da una classe (non esiste ereditarietà multipla)
- Se non viene specificata una superclasse, implicitamente da Object
  - Object è l'unica classe a non avere superclasse



Ereditarietà

- L'ereditarietà permette di creare gerarchie di classi che includono campi e metodi comuni
  - Il costruttore non viene mai ereditato, ma può essere acceduto

```
public class Square extends Rectangle {

   public Square(int side) {
      super(side, side)
   }

}
```

Ereditarietà
Visibilità

- Una sottoclasse eredita:
  - tutti i campi e metodi dichiarati **public** e **protected**
  - tutti i campi **package-private**, se la superclasse è nello stesso package
  - nessun campo o metodo **private**
- Tutti i campi e i metodi ereditati, possono essere usati **direttamente** nella sottoclasse

```
Square square = new Square(5);

// inherited from Rectangle (actually, from Shape)
square.getArea();
```

Ereditarietà
Visibilità

- In una sottoclasse si possono dichiarare nuovi campi
- Se si dichiara un campo con lo stesso nome di un campo presente nella superclasse, allora si sta "nascondendo" tale campo (sconsigliato)
  - il campo della superclasse non sarà più direttamente accessibile

Ereditarietà
Visibilità

- Se un campo è dichiarato come private (per l'incapsulamento), allora una sottoclasse non può accedervi direttamente
- Se però è stato fornito un metodo di accesso (getter/setter), allora la sottoclasse può accedere al campo

Ereditarietà
Visibilità

- In una sottoclasse si possono dichiarare nuovi metodi
- Se si dichiara un metodo con la stessa firma *(i.e. nome, parametri e tipo di ritorno)* di un metodo nella superclasse, allora si sta facendo l'**override** di tale metodo
  - Se il metodo è statico, si sta nascondendo (non ha senso override di metodi statici)

Override

```
public class Rectangle implements Shape {

   ...

   public String getIdentifier() {
     return “rect”;
   }

}
```

```
public class Square extends Rectangle {

   ...

   @Override
   public String getIdentifier() {
     return “square”;
   }

}
```

Override

- Il metodo che fa override deve avere
  - lo stesso nome
  - lo stesso numero, tipo e ordine di parametri
  - lo stesso tipo di ritorno
    - Il tipo di ritorno può essere anche una **sottoclasse** del tipo di ritorno del metodo sovrascritto
- L'annotazione **@Override** viene usata per indicare al compilatore la volontà di fare override
  - Non è obbligatorio, ma serve per evitare eventuali errori

Override

- Quando si fa override, si può dare più visibilità
- Non è possibile invece dichiarare metodi meno visibili

```
public class Rectangle implements Shape {
   ...
   protected String getIdentifier() {
      return “rect”;
   }
}
```

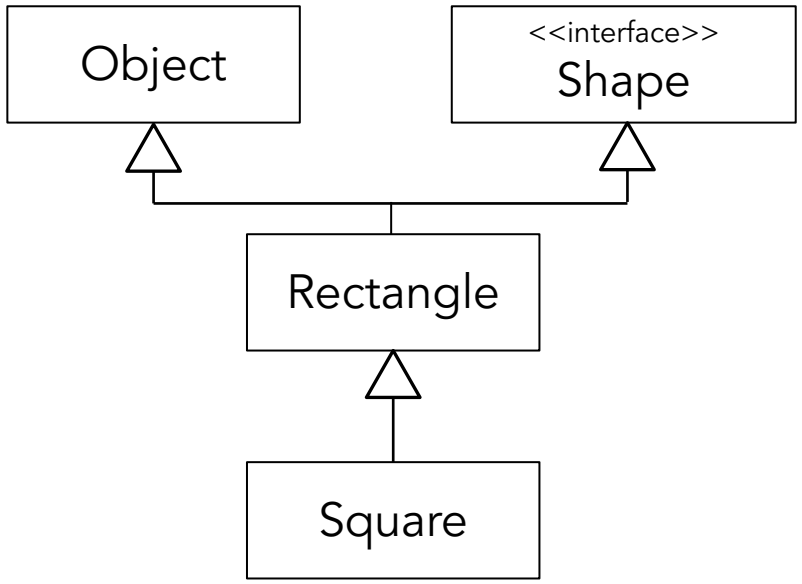
```
public class Square extends Rectangle {
   ...
   @Override
   public String getIdentifier() {
      return “square”;
   }
}
```

```
public class Rectangle implements Shape {
   ...
   public String getIdentifier() {
      return “rect”;
   }
}
```

```
public class Square extends Rectangle {
   ...
   @Override
   protected String getIdentifier() {
      return “square”;
   }
}
```

Cast

- Il cast permette di forzare il tipo di una variabile ad un altro, a patto che i tipi appartengano alla stessa gerarchia



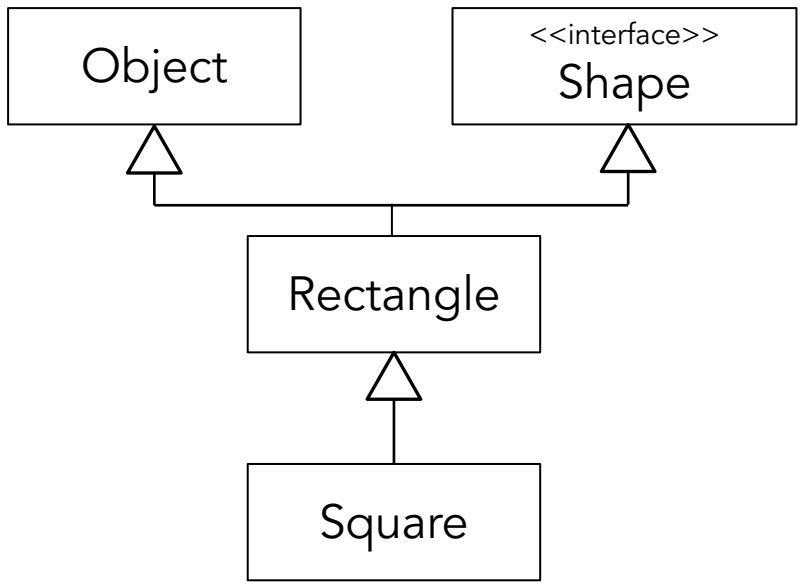
Cast

- implicito

```
Object o = new Rectangle();
Shape s = new Rectangle();
Shape s = new Square();
Rectangle r = new Square();
```

- esplicito

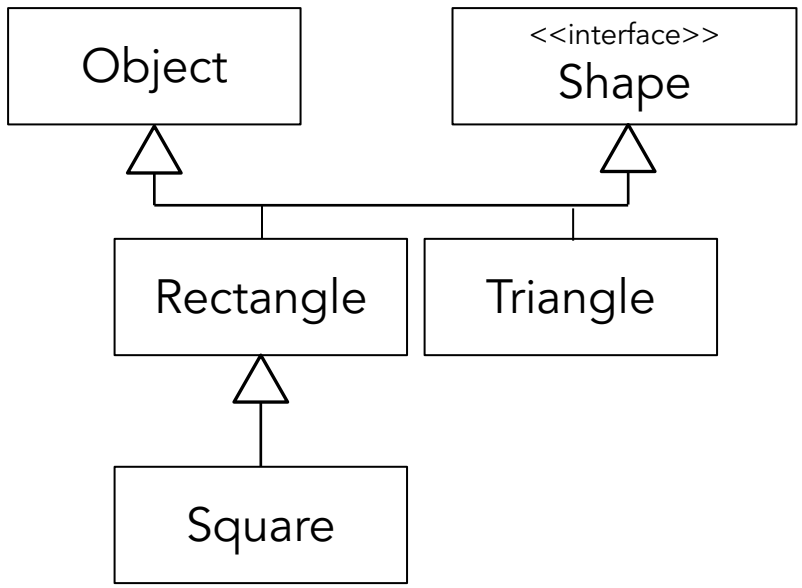
```
Object o = new Square();
Shape s = new Rectangle();

Rectangle r = s;
Rectangle r = (Rectangle)s;

Square s = o;
Square s = (Square)o;
```



Cast ClassCastException

```
Triangle t = (Triangle) new Rectangle();

Shape s = new Rectangle();
Triangle t = (Triangle) s;
```

- Il secondo caso per il compliatore è corretto…
- …ma a runtime viene sollevata un'eccezione!
  - ClassCastException



Cast
instance of

- Si può testare la classe di un'istanza prima di fare un cast, in modo da non avere mai problemi di cast

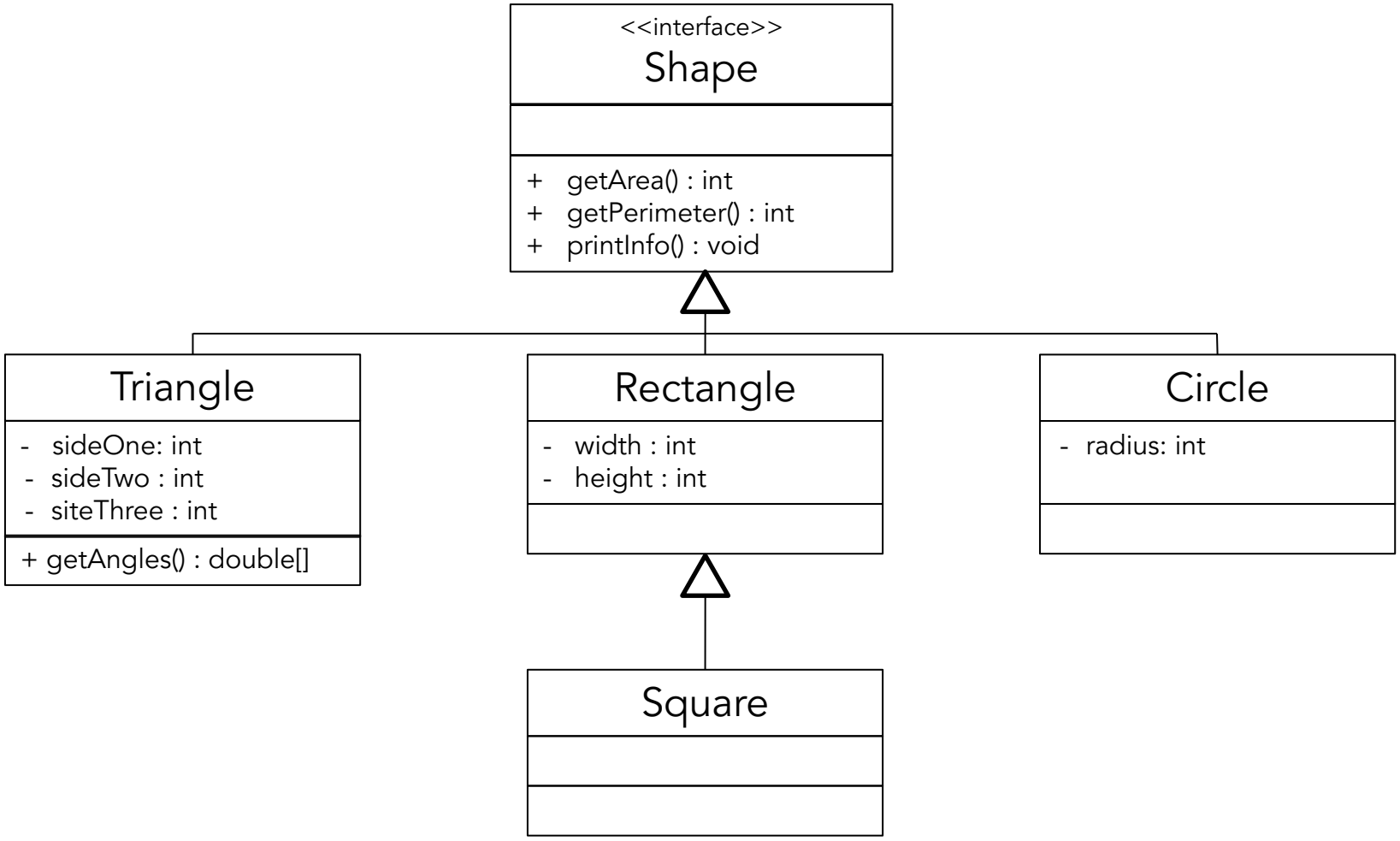
```
Shape s = new Rectangle();

if(s instanceof Triangle) {
    Triangle t = (Triangle) s;
}
```

- Esempio: iterare su un array di forme e stampare un messaggio diverso a seconda del tipo

Polimorfismo

- Polimorfismo = capacità di avere più forme
- In OOP, il polimorfismo si riferisce alla capacità di una oggetto di essere istanza di più tipi (classi e/o interfacce)
- Questo apre la strada alla scrittura di codice che opera su tipi più o meno generali, in modo da ottenere il giusto grado di astrazione e di disaccoppiamento

Polimorfismo



super

- Permette di accedere a metodi e campi della superclasse

```
public int aMethod(int value) {
    return super.aMethod(value)
}
```

- Viene usato per invocare un costruttore della classe padre

```
public class Square extends Rectangle {

   public Square(int side) {
      super(side, side)
   }

}
```

final

- Tramite la keyword final è possibile:
  - definire un metodo di cui non può essere fatto l'override

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
     return ChessPlayer.WHITE;
    }
    ...
}
```

  - definire una classe che non può essere estesa da nessuna sottoclasse
    - utile per classi immutabili (e.g. String)

Classi e metodi astratti

- Una classe o un metodo possono essere astratte
  - Un metodo astratto viene dichiarato senza implementazione
    - Deve essere contenuto in una classe astratta
    - I metodi delle interfacce sono implicitamente astratti
  - Una classe astratta non può essere istanziata, ma può essere estesa
    - Quando viene estesa deve fornire un'implementazione per tutti i metodi astratti dichiarati nella classe

Classi astratte vs Interfacce

- Similarità
  - Non possono essere istanziate
  - Possono contenere metodi senza o con implementazione
- Differenze
  - Nelle classi astratte si possono dichiarare campi
  - Nelle interfacce si possono definire solo metodi pubblici
  - Si può estendere solo una classe astratta, ma si possono implementare più di un'interfaccia

Classi astratte vs Interfacce

- Situazioni in cui usare **classi astratte**
  - quando si vuole condividere codice fra più classi che hanno lo stesso dominio
  - quando si vuole definire una classe da estendere, che comprende campi e metodi non pubblici
- Situazioni in cui usare **interfacce**
  - quando si vuole definire un comportamento implementabile da classi di dominio diverso
  - quando si vuole sfruttare l'ereditarietà multiple

Interfacce
Default Methods

- Quando una classe estende un interfaccia che contiene un **default method**, si può:
  - non farci niente: si eredita il metodo e la sua implementazione di default
  - dichiarare di nuovo il metodo: in questo caso va definito come abstract
  - dare una nuova implementazione: viene fatto l'override

Nested Classes

Nested Classes

- Classe definita all'interno di un'altra classe
  - E' un campo della classe all'interno della quale è stata dichiarata
  - Può essere dichiarata private, public, protected o package private
  - Accede a tutti i campi della classe che la racchiude, anche privati
  - Può essere anche statica (accede solo ai campi statici)

Nested Classes Sintassi

```
public class OuterClass {
     //...
     static class StaticNestedClass {
          //...
     }
     class InnerClass {
          //...
     }
}
```

- Terminologia:
  - Classe statica: *Static Nested Class*
  - Classe non statica: *Inner Class*

Nested Classes Motivazioni

- Benefici nell'uso delle classi annidate:
  - Raggruppamento logico di classi che sono usate solo in un posto
  - Aumento dell'incapsulamento
  - Scrittura di codice più manutenibile e leggibile

Nested Classes Esempio

```
public class Rectangle {

    //...

    public boolean isBigger(Rectangle other) {
        return new RectangleComparator().compare(this, other) > 0;
    }

    private static class RectangleComparator implements Comparator<Rectangle>
    {
        @Override
        public int compare(Rectangle r1, Rectangle r2) {
              return r1.getArea() - r2.getArea();
        }
    }

}
```

Local Classes

- Classi che sono definite all'interno di un block (tipicamente il corpo di un metodo)
- Ha accesso a tutti i campi della classe che la racchiude…
- …e anche alle variabili locali e i parametri del blocco, a patto che siano final o effective final

Local Classes Esempio

```
public class Rectangle {

    //...

    public boolean isBigger(Rectangle other) {
        class RectangleComparator implements Comparator<Rectangle> {
            @Override
            public int compare(Rectangle r1, Rectangle r2) {
                  return r1.getArea() - r2.getArea();
            }
        }

    return new RectangleComparator().compare(this, other) > 0;
    }

}
```

Anonymous Classes

- Classi dichiarate all'interno di espressioni
- Sono loro stesse **espressioni**, composte da
  - operatore new
  - il nome di un'interfaccia da implementare, o una classe da estendere
  - argomenti del costruttore, passati tra parentesi
  - corpo del metodo, che contiene l'implementazione
- Seguono le stesse regole delle local classes per quanto riguarda l'utilizzo dei campi e variabili

Anonymous Classes Esempio

```
public class Rectangle {

    //...

    public boolean isBigger(Rectangle other) {
        return new Comparator<Rectangle>() {
            @Override
            public int compare(Rectangle r1, Rectangle r2) {
                  return r1.getArea() - r2.getArea();
            }
        }.compare(this, other) > 0;

    }

}
```

Esercizio

Scrivere una classe che riceve in ingresso sequenzialmente una serie di numeri, lo filtra secondo una strategia impostabile (es: numero > 0, numero pari) e restituisce l'insieme di numeri filtrati.

Object

Object come superclasse

- Tutte le classi in Java ereditano, direttamente o indirettamente, dalla classe Object
- Da Object, vengono quindi ereditati implicitamente alcuni metodi
  - Possono essere visti su Eclipse quando si autocompleta i metodi di una classe dopo il “.”

Object come superclasse

- I più interessanti (ed importanti) di questi metodi sono
  - public boolean **equals**(Object obj)
    - Indica se un certo oggetto è "uguale" a questo
  - public int **hashCode**()
    - Ritorna l'hashCode di questo oggetto
  - public String **toString**()
    - Ritorna una rappresentazione a stringa dell'oggetto
  - public final Class **getClass**()
    - Ritorna la classe dell'oggetto

equals
Identità vs uguaglianza

- Identità e uguaglianza sono due concetti distinti e a cui va prestato attenzione
  - Prendiamo la classe String come esempio

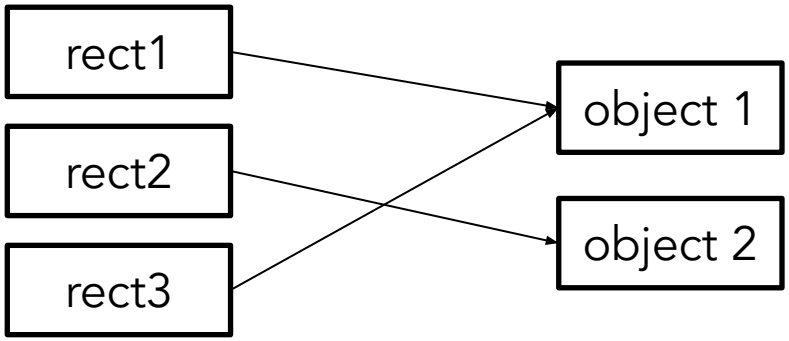
```
String string1 = new String("pippo");
String string2 = new String("pippo");
System.out.println( string1 == string2 );

System.out.println( string1.equals(string2 ) );
System.out.println( string2.equals(string1 ) );
```

Identità

- Quando si dichiara e istanzia un oggetto, viene riservato uno spazio in memoria per salvarlo
- L'operatore == applicato agli oggetti testa l'identità, ovvero se due oggetti **sono lo stesso** (occupano lo stesso spazio in memoria)
  - L'implementazione di base di equals è equivalente a testare l'identità

Identità

```
Rectangle rect1 = new Rectangle(3, 4);
Rectangle rect2 = new Rectangle(5, 2);
Rectangle rect3 = rect1;

System.out.println( rect1 == rect2 );
System.out.println( rect2 == rect3 );
System.out.println( rect1 == rect3 );
```



Uguaglianza

- Il metodo **equals** testa l'uguaglianza, ovvero se due oggetti sono considerati uguali
  - Per una classe come **String** è facile immaginare cosa vuol dire: sono uguali se hanno gli stessi caratteri nello stesso ordine
  - Per un oggetto come il nostro Rectangle, cosa significa?

Uguaglianza

- Spetta a chi implementa la classe definire l'uguaglianza per istanze di quella classe
  - Nel nostro caso, portrebbe essere: *"due rettangoli sono uguali se hanno i lati uguali"*

```
Rectangle rect1 = new Rectangle(3, 4);
Rectangle rect2 = new Rectangle(5, 2);
Rectangle rect3 = new Rectangle(3, 4);

System.out.println( rect1.equals(rect2 ) ); //false
System.out.println( rect1.equals(rect3)  ); //true
System.out.println( rect2.equals(rect3) ); //false
```

Uguaglianza

- Per implementare il controllo di uguaglianza per una classe, occorre fare l'**override** del metodo **equals()**
- Esistono delle linee guida che Java invita a seguire per una corretta implementazione del metodo **equals()**
  - Può essere trovato (in inglese) sulla javadoc

Object.equals

- public boolean equals(Object obj)
- Indicates whether some other object is "equal to" this one.
- The equals method implements an equivalence relation on non-null object references:
  - It is *reflexive*
    - for any non-null reference value x, x.equals(x) should return true

Object.equals

- It is *symmetric*:
  - for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- It is *transitive*:
  - for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true
- It is *consistent*:
  - for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified

Object.equals

- For any non-null reference value x, x.equals(null) should return false
- The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true)
- Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes

Object.equals quando?

- Quando è consigliabile fare override di equals?
- Quando la classe è una classe che:
  - ha una nozione di uguaglianza diversa da quella di identità
  - non eredita l'implementazione di equals da una superclasse

Object.equals come?

- testare l'identità con l'operatore ==
  - non necessario, ma alleggerisce l'algoritmo
- usare instanceof per controllare la classe dell'altro oggetto
  - usare la classe dell'oggetto, o quella dell'interfaccia se si vuole permettere comparazioni trasversali
- fare il cast alla classe giusta

da Effective Java, di Joshua Bloch

Object.equals come?

- testare l'uguaglianza per ogni campo significativo della classe
  - se tutti i controlli hanno successo, ritorna true
  - appena un controllo fallisce, ritorna false
  - per i campi di tipo primitivo, si utilizza == (tranne con float e double!)
  - se i campi sono istanze di una classe, utilizzare il metodo equals di quella classe
  - prevenire le NullPointerException

da Effective Java, di Joshua Bloch

Object.equals come?

- Controllare se rispetta i requisiti
  - è riflessivo?
  - è simmetrico?
  - è transitivio?
  - è consistente?

- Proseguire facendo l'**override** del metodo hashCode()

da Effective Java, di Joshua Bloch

Object.hashCode

- Funzione di hash: funzione che può essere usata per mappare dati di grandezza arbitraria in dati di grandezza fissata
  - Input: dato di qualsiasi genere
  - Output: codice hash
- Esempi di utilizzo:
  - trovare corrispondenza tra due input uguali senza conoscere l'input (sicurezza per password)
  - controllare presenza duplicati in modo immediato

Object.hashCode

- public int hashCode()
- Returns a hash code value for the object
- The general contract of hashCode is:
  - Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified.
    - This integer need not remain consistent from one execution of an application to another execution of the same application.

Object.hashCode

- If two objects are equal according to the **equals(Object)** method, then calling the hashCode method on each of the two objects must produce the same integer result

- It is *not* required that if two objects are unequal according to the **equals(Object)** method, then calling the **hashCode** method on each of the two objects must produce distinct integer results.

  - However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables

Object.hashCode quando?

- Ogni volta che si fa l'**ovveride** di **equals**
- Esistono diverse strutture dati in Java che utilizzano l'hashCode per funzionare
- Se viene cambiata l'implementazione di **equals** senza cambiare quella di **hashCode**, la classe non funzionerà più correttamente quando si andranno ad usare queste strutture dati
  - L'implementazione di default ritorna come hashCode la rappresentazione intera dell'indirizzo di memoria in cui è contenuto l'oggetto

Object.hashCode come?

- Creare una funzione di hash che dia buoni risultati può essere complicato
- Una buona tecnica è:
  - scegliere una costante diversa da 0 (e.g. 17)
  - per ogni campo utilizzato nel calcolo di equals
    - calcolare l'hashCode del campo
    - combinarlo col risultato moltiplicato per una costante (e.g. 31) e sommarlo

```
int result = 17;
result = 31 * result + c;
…
```

da Effective Java, di Joshua Bloch

Object.hashCode come?

- Per i tipi primitivi, calcolare l'hash dipende dal tipo
  - se f è di tipo **byte**, **char**, **short**, or **int**, calcola (int)f
  - se è boolean, calcola (f ? 1: 0)
- Se il tipo è un istanza di una classe, utilizzare la funzione **hashCode** di quella classe
  - Controllare sempre se l'oggetto è nullo, ed eventualmente tornare una costante (e.g. 0)

da Effective Java, di Joshua Bloch

Object.toString

- Il metodo **toString** fornisce una rappresentazione a stringa di un oggetto
- L'implementazione di default ritorna una rappresentazione nel formato

```
<nomeClasse>@<hashCode>
```

- Dare una rappresentazione più significativa può facilitare operazioni di debug e log