Generics I

- The original Collections framework just dealt with collections of **Object**s
  - Everything in Java “is-a” **Object** so that way our collections framework will apply to any class we like without any special modification.
  - It gets messy when we get something from our collection though: it is returned as an **Object** and we have to do a narrowing conversion to make use of it:

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
      Object o = it.next();
      Integer i = (Integer)o;
}
```

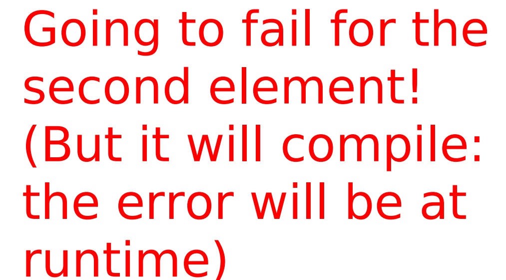
Generics II

- It gets worse when you realise that the add() method doesn't stop us from throwing in random objects:

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person("Bob"));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
      Object o = it.next();
      Integer i = (Integer)o;
}
```



Generics III

- To help solve this sort of problem, Java introduced *Generics* in JDK 1.5
- Basically, this allows us to tell the compiler what is supposed to go in the Collection
- So it can generate an error at compile-time, not run-time

```
// Make a TreeSet of Integers
TreeSet<Integer> ts = new TreeSet<Integer>();

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person("Bob"));

// Loop through
iterator<Integer> it = ts.iterator();
while(it.hasNext()) {
      Integer i = it.next();
}
```

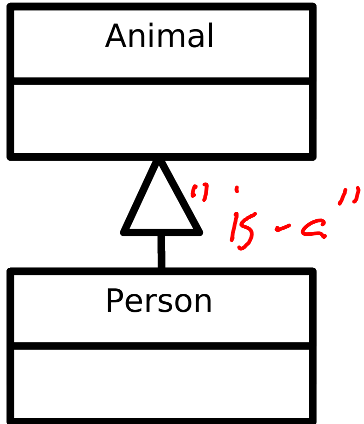
Won't even compile

No need to cast :-)

Notation in Java API

- Set<E>
- List<E>
- Queue<E>
- Map<K,V>

Generics and SubTyping




```
// Object casting
Person p = new Person();
Animal o = (Animal) p;

// List casting
List<Person> plist = new LinkedList<Person>();
List<Animal> alist = (List<Animal>)plist;
```

So a list of **Person**s is a list of **Animal**s, yes?

Comparing Java Classes

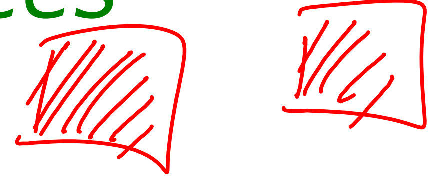
Comparing Primitives

>	Greater Than
>=	Greater than or equal to
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to

- Clearly compare the value of a primitive
- But what does (object1==object2) mean??
  - Same object?
  - Same state (“value”) but different object?

Option 1: a==b, a!=b

- These compare the *references*



```
Person p1 = new Person("Bob");
Person p2 = new Person("Bob");

(p1==p2);

(p1!=p2);

p1==p1;
```

False (references differ)

True (references differ)

True (references the same)

```
String s = "Hello";
if (s=="Hello") System.out.println("Hello");
else System.out.println("Nope");
```

Option 2: The equals() Method

- Object defines an equals() method. By default, this method just does the same as ==.
  - Returns boolean, so can only test equality
  - Override it if you want it to do something different
  - Most (all?) of the core Java classes have properly implemented equals() methods

```
Person p1 = new Person("Bob");
Person p2 = new Person("Bob");

(p1==p2);

String s1 = "Bob";
String s2 = "Bob";

(s1==s2);
```

p1.equals(p2)

s1.equals(s2)

False (we haven't overridden the equals() method so it just compares references

True (String has equals() overridden)

Option 3: Comparable<T> Interface

<T>

```
int compareTo(T obj);
```

- Part of the Collections Framework
- Returns an integer, r:
  - r<0 This object is less than obj
  - r==0 This object is equal to obj
  - r>0 This object is greater than obj

Natural ordering

Option 3: Comparable<T> Interface II

```
public class Point  implements Comparable<Point> {
     private final int mX;
     private final int mY;
     public Point (int, int y) { mX=x; mY=y; }

     // sort by y, then x
     public int compareTo(Point p) {
         if ( mY>p.mY) return 1;
         else if (mY<p.mY) return -1;
         else {
             if (mX>p.mX) return 1;
             else if (mX<p.mX) return -1;
             else return 0.
         }
     }
}
```

```
// This will be sorted automatically by y, then x
Set<Point> list = new TreeSet<Point>();
```

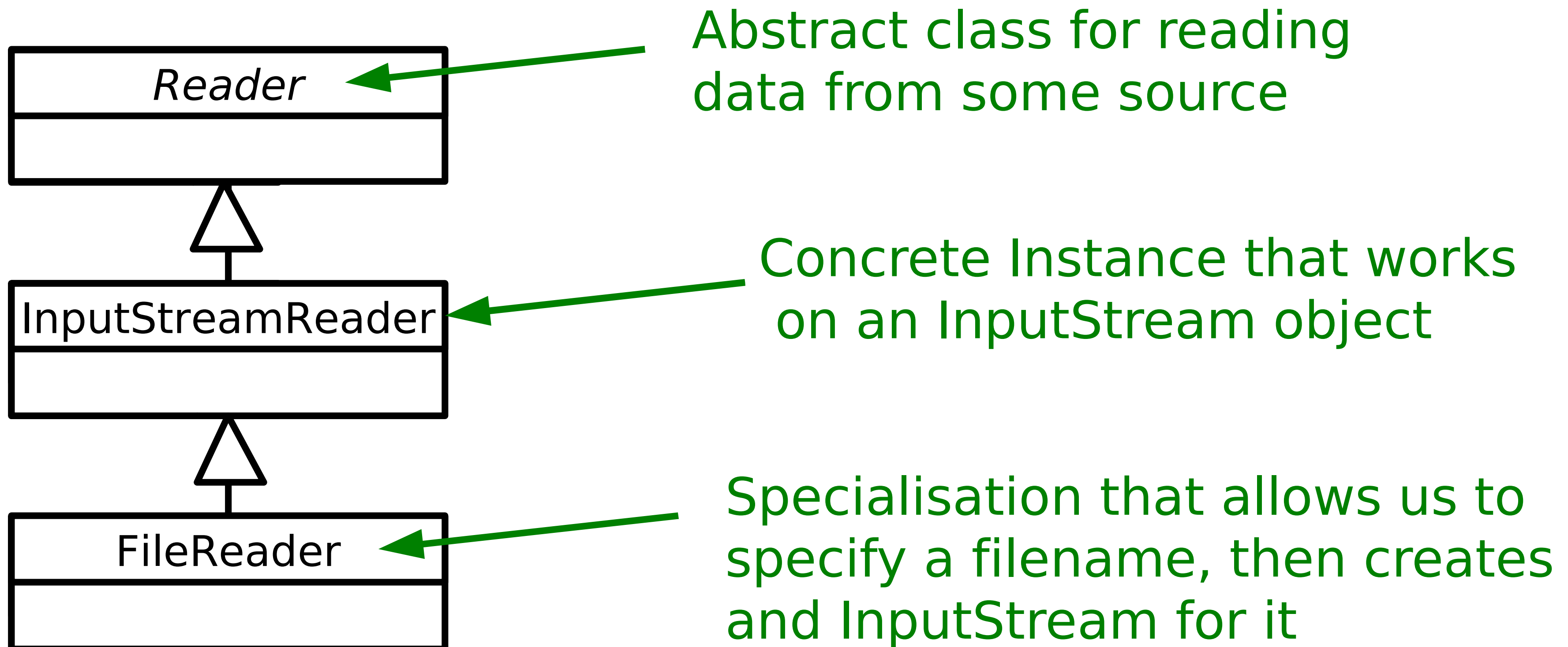
Option 4: Comparator<T> Interface

int compareTo(T obj1, T obj2)

- Also part of the Collections framework and allows us to specify a particular comparator for a particular job
- E.g. a Person might have a compareTo() method that sorts by surname. We might wish to create a class AgeComparator that sorts Person objects by age. We could then feed that to a Collections object.

Java's I/O framework

- Support for system input and output (from/to sources such as network, files, etc).



Speeding it up

- In general file I/O is sloowwww
- One trick we can use is that whenever we're asked to read some data in (say one byte) we actually read lots more in (say a kilobyte) and buffer it somewhere on the assumption that it will be wanted eventually and it will just be there in memory, waiting for us. :-)
- Java supports this in the form of a **BufferedReader**

```
FileReader f = new FileReader();
BufferedReader br = new BufferedReader(f);
```

- Whenever we call read() on a BufferedReader it looks in its buffer to see whether it has the data already
- If not it passes the request onto the Reader object
- We'll come back to this...