CSC 7322 : Object Oriented Development

J Paul Gibson, A207

paul.gibson@int-edu.eu

http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7322/

Advanced Concepts (in Java)

…/~gibson/Teaching/CSC7322/L4-AdvancedConcepts.pdf

Some More Advanced Concepts

1. Generics
2. Exceptions
3. Threads
4. Packages
5. Serialization
6. Input/Output
7. Files
8. Reflection
9. Annotations
10. Design Patterns (in following lecture(s))

NOTE: We will look at each of these as the need arises (or as you request).

QUESTION - Which of these are:

- orthogonal to OO, or
- have an OO flavour, or
- are fundamentally OO

1 Generics - Some History

M.D. McIlroy: *Mass-Produced Software Components*, **Proceedings of the 1st International Conference on Software Engineering**, Garmisch Pattenkirchen, Germany, **1968**

Joseph A. Goguen: *Parameterized Programming*. **IEEE Trans. Software Eng. 10(5) 1984**

David R. Musser, Alexander A. Stepanov: *Generic Programming*. **ISSAC 1988**

Charles W Kreuger, *Software Reuse*, **ACM Computing Surveys, 1992**

Ronald Garcia et al, *A Comparative Study of Language Support for Generic Programming*, **OOPSLA03, 2003**

1 Generics - Some Java History

Martin Odersky and **Philip Wadler**. *Pizza into Java: translating theory into practice*. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (**POPL '97**).

Pizza

Gilad Bracha, Martin Odersky, David Stoutamire, and **Philip Wadler**. *Making the future safe for the past: adding genericity to the Java programming language*. ***SIGPLAN Not.*** **33, 10 (October 1998)**,

GJ

May 1999 - Sun proposes to Add Generics to Java, based on GJ. The activity (named JSR 14) is headed by Gilad Bracha

JSR-000014 Adding Generics to the Java™ Programming Language
(Close of Public Review: 01 August 2001)
http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html

JSR-000014

Mads Torgersen et al., *Adding wildcards to the Java programming language*, Proceedings of the **2004 ACM symposium on Applied computing**.

JDK1.5

Recently : push for simplifying/eliminating wildcards!!!

1 A Generic Swap: in different languages

```
template <class T>
void swap( T& a, T& b){
T tmp = a;
a = b;
b = tmp;
}
```

```
sub swap {@_[0, 1] = @_[1, 0]}
```

```
static void Swap<T>(ref T a, ref T b){
T temp = a;
a = b;
b = temp;
}
```

```
let swap (a,b) = (b,a)
```

QUESTION: do you recognise any of these?

1 A Generic Swap: in different languages

```
generic
type Swap_Type is private; -- Generic parameter
procedure Generic_Swap(Left : in out Swap_Type; Right : in out Swap_Type);

procedure Generic_Swap(Left : in out Swap_Type; Right : in out Swap_Type) is
Temp : Swap_Type := Left;
begin
Left := Right;
Right := Temp;
end Generic_Swap;
```

```
class Pair<T> {
T first;
T second;

public static <T> void swap(Pair<T> p) {
T temp = p.first;
p.first = p.second;
p.second = temp;
}}
```

1 Why are generics useful

Re-usable patterns (like higher order functions):

foldl (+) 0 [1..5] = 15
foldl (append) "" ["a", "b", "c"] = " abc"

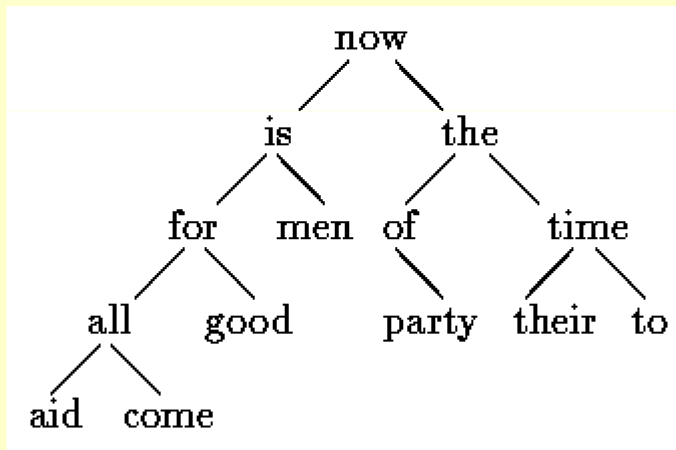
filter (odd) [1,3,5,2,4] = [1,3,5]
filter (animal) [cow, dog, cake] = [cow, dog]

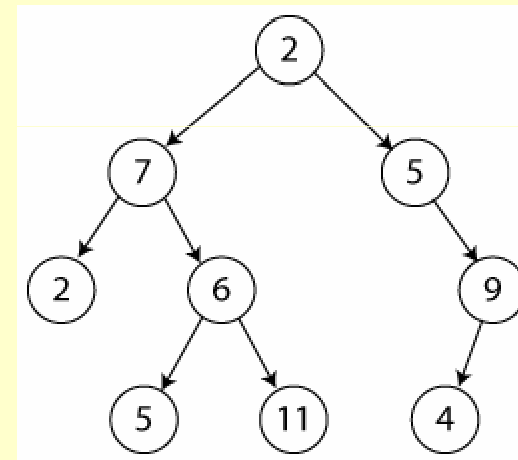
map (double) [1,3,5,2,4] = [2,6,10,4,8]
map (capitalize) ["aBc", "BBc"] = ["ABC", "BBC"]

QUESTION: what are the types of these 3 functions?

1 Why are generics useful

Re-usable data structures, eg binary tree of things:





With generic algorithms/functions, eg depth

1 Why are generics useful

Re-usable classes, eg (ordered) list of things:

- Combines generic data and generic functions in generic class
- **Unconstrained genericity** – no restriction on type/class of generic parameter
- **Constrained genericity** – the generic parameter must be a type/class which is a subtype/subclass of a specified class

NOTE: Genericity is usually extended to allow multiple generic parameters (but then they may/may not be mutually constrained)

1 Why are generics useful: a classic Java example

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

QUESTION: Why do we not like this type of code?

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

QUESTION: Why do we prefer this type of code?

NOTE: The 2nd example uses the Java **List** collections class

Why are generics useful: Java List example, continued:

```
public interface List<E> {
void add(E x);
Iterator<E> iterator();
}
```

The declaration of the *formal type parameters of the interface List*

You might imagine that List<Integer> stands for a version of List where E has been uniformly replaced by Integer:

```
public interface IntegerList {
void add(Integer x)
Iterator<Integer> iterator();
}
```

This intuition may be useful, but it may also be misleading. (This is closer to the type of macro expansion in the C++ STL)

Java generics implemented by erasure

Generics are implemented by the Java compiler as a front-end conversion called *erasure. You can (almost) think of it as a source-to-source* translation (syntactic sugar), whereby the generic version of code is converted to the non-generic version.

As a result, **the type safety and integrity of the Java virtual machine are never at risk, even in the presence of unchecked warnings.**

Basically, erasure gets rid of (or *erases) all generic type information. All the type* information between angle brackets is thrown out, so, for example, a parameterized type like List<String> is converted into List. All remaining uses of type variables are replaced by the upper bound of the type variable (usually Object). And, whenever the resulting code isn't type-correct, a cast to the appropriate type is inserted.

How To Implement Generics – many choices (see referenced papers)

While generics look like the C++ templates, it is important to note that they are not (implemented) the same.

Java generics simply provide compile-time type safety and eliminate the need for casts.

Generics use a technique known as type erasure as described above, and the compiler keeps track of the generics internally, and all instances use the same class file at compile/run time.

A C++ template on the other hand is just a fancy macro processor; whenever a template class is instantiated with a new class, the entire code for the class is reproduced and recompiled for the new class.

Some Java "Details" : all instances of a generic class have the same run-time class

What does the following code fragment print?

```
List <String> l1 = new ArrayList<String>();
List <Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

It prints true, because all instances of a generic class have the same run-time class, regardless of their actual type parameters.

As consequence, the static variables and methods of a class are also shared among all the instances. That is why it is illegal to refer to the type parameters of a type declaration in a static method or initializer, or in the declaration or initializer of a static variable.

Generics and Subtyping

QUESTION: What does the following code output?

```
public class InheritanceTester {

public static void main(String[] args) {
 List<Dog> animals = new ArrayList<Dog>();
 message(animals);
}
 private static void message(Collection<Animal> animals) {
System.out.println("You gave me a collection of animals.");
}

private static void message(Object object) {
System.out.println("You gave me an object.");
}
}
```

In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is **not the case that G<Foo> is a subtype of G<Bar>.**

This is probably the hardest thing you need to learn about (Java) generics … and how it relates to the concept of wildcards

Generics and Subtyping   Example: drawing shapes in a canvas

Typically , a drawing will contain a number of shapes.

Assuming that the shapes are stored in a list, it would be convenient to have a method in Canvas that draws them all:

```
public void drawAll(List<Shape> shapes) {
 for (Shape s: shapes) { s.draw(this); }
}
```

Now, the type rules (as we saw on previous slide) say that drawAll() can only be called on lists of exactly Shape: it cannot, for instance, be called on a List<Circle>.

That is unfortunate, since all the method does is read shapes from the list, so it could just as well be called on a List<Circle>…

Java wildcards were introduced to overcome this problem.

Wildcards – drawing shapes in a canvas

What we really want is for the method to accept a list of **any** kind of shape:

```
public void drawAll(List<? extends Shape> shapes) {
... }
```

There is a small but very important difference here: we have replaced the type List<Shape> with List<? **extends** Shape>.

Now drawAll() will accept lists of any subclass of Shape (or Shape itself), so we can now call it on a List<Circle> if we want.

List<? **extends** Shape> is an example of a *bounded wildcard*.

We say that Shape is the *upper bound* of the wildcard.

Java Wildcards

There are three types of wildcards in Java:

1. "? extends Type": Denotes a family of subtypes of type Type. This is the most useful wildcard

2. "? super Type": Denotes a family of supertypes of type Type.

3. "?": Denotes the set of all types or *any*

Question: can you think of a use of the second wildcard type?

Problem: Implement a Pair Of *Things* in Java

You are to code the class `GenericPair`, such that it passes the tests written in `JUnit_GenericPairTest` (which can be downloaded from the module web site).

templates.GenericPair<T>

A 2-tuple (pair) of *things*

For teaching advanced OO concepts in Java - genericity

Parameters:
<T> is the type/class of the pair

Version:
1

Author:
J Paul Gibson

tests.JUnit_GenericPairTest

A test for a 2-tuple (pair) of *things*

For teaching advanced OO concepts in Java - genericity

Uses **JUnit** to test template class `GenericPair`

Version:
1

Author:
J Paul Gibson

NOTE: The code in GenericPair.java needs to be completed

Problem: Implement a Pair Of *Things* in Java

Field Summary

(package private) GenericPair<java.lang.Character>	**poc** A pair of characters
(package private) GenericPair<java.lang.Character>	**poc_copy** A copy of the pair of Characters `poc`
(package private) GenericPair<java.lang.Integer>	**poi** A pair of integers
(package private) GenericPair<java.lang.Integer>	**poi_copy** A copy of the pair of Integers `poi`
(package private) GenericPair<GenericPair<?>>	**pop** A generic pair of pairs
(package private) GenericPair<GenericPair<java.lang.Character>>	**popoc** A pair of a pair of Characters
(package private) GenericPair<GenericPair<java.lang.Integer>>	**popoi** A pair of a pair of Integers

The test variables

setUp

Initialise the test variables

- poi and poi_copy as pair of Integers (0,0)
- poc and poc_copy as a pair of Characters ('a', 'b')
- popoi as (('1','2') , ('3','4')
- popoc as (('a','b') , ('c','d'))
- pop as (('1','2') , ('c','d'))

Problem: Implement a Pair Of *Things* in Java

The tests:

`testToString`
Tests method `GenericPair.toString()`

`testSwap_static`
Tests method `GenericPair.swap(GenericPair)`

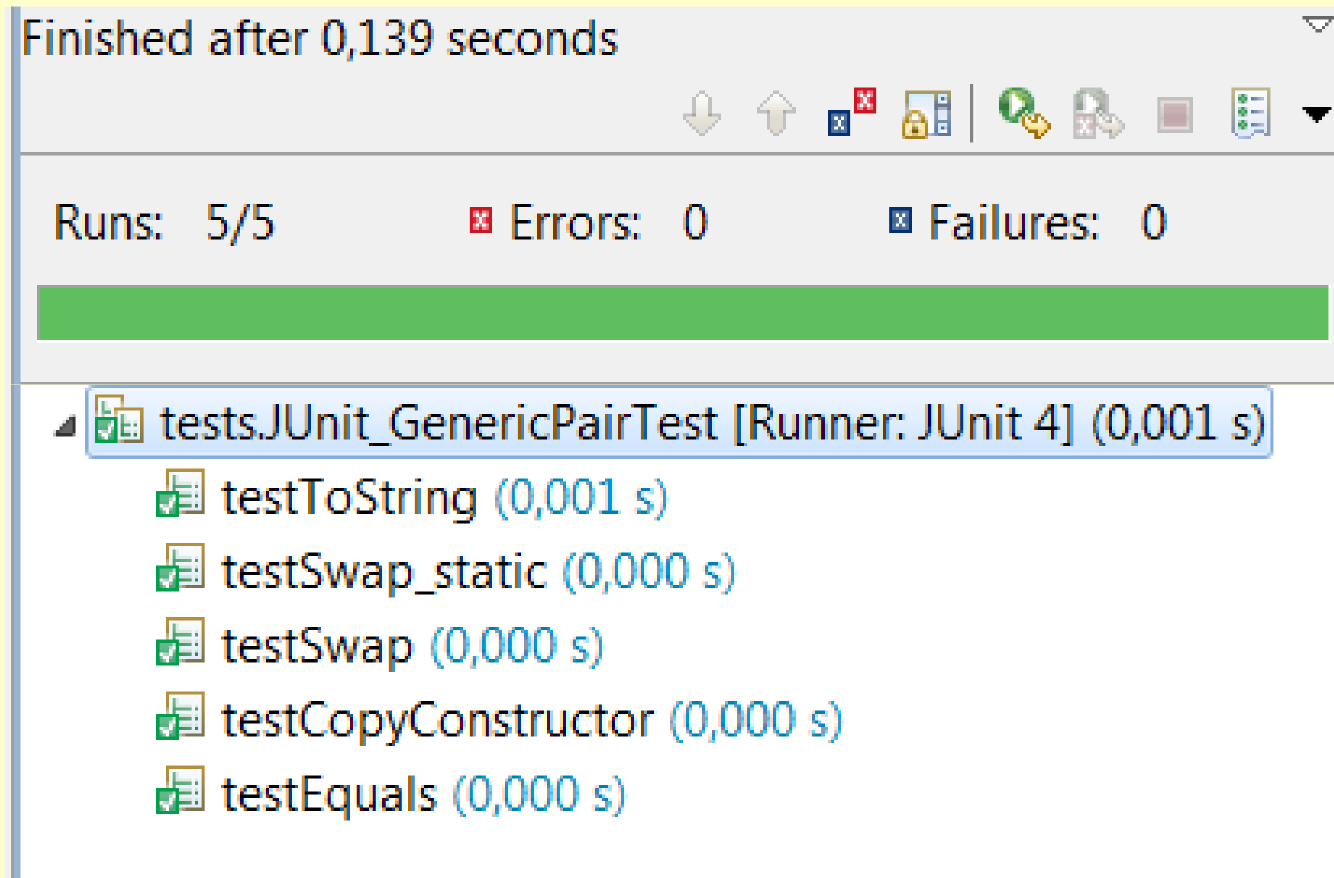
`testSwap`
Tests method `GenericPair.swap()`

`testCopyConstructor`
Tests method `GenericPair.GenericPair(GenericPair)`

`testEquals`
Tests method `GenericPair.equals(java.lang.Object)`

Problem: Implement a Pair Of *Things* in Java

TO DO: Write the `GenericPair` so that the tests are successful



Problem: Implement a Pair Of *Things* in Java

TO DO: Write the `GenericPair` so that the tests are successful

You should consider the test code to specify the requirements.

For example, you can deduce that you need constructors:

templates.GenericPair.GenericPair(T first, T second)

Explicit constructor

Parameters:
first is the initial value of the first element
second is the initial value of the second element

templates.GenericPair.GenericPair(GenericPair<T> pair)

Shallow copy constructor, where first and second values are copied by reference

Parameters:
pair is the pair to be copied

Problem: Implement a Pair Of *Things* in Java (using generics)

TO DO: Write the `GenericPair` so that the tests are successful

For example, you can also deduce that you need 2 swap methods:

void templates.GenericPair.swap(GenericPair<T> p)

Parameters:

<T> the type of pair elements to be swapped

p the pair to be swapped

void templates.GenericPair.swap()

Swap the first and second values of the pair

QUESTION: What other methods do you need?

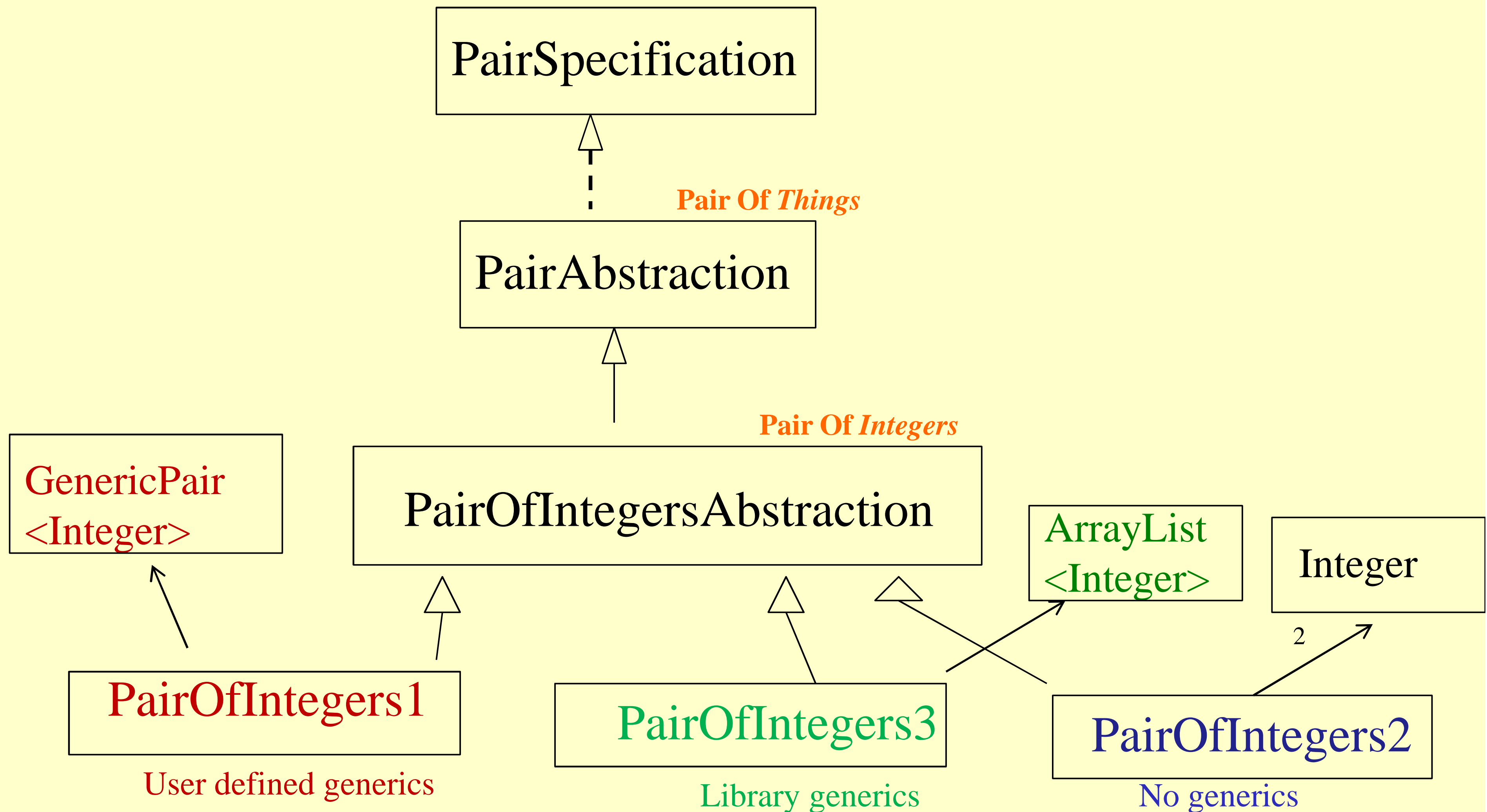
Problem: Implement a Pair Of *Things* in Java (without generics)

Let us consider how we could do this **without** the generic templates

One such approach is to use the base class `Object`

```
public interface PairSpecification {

Public void swap();
public Object getFirst();
public Object getSecond();
public void setFirst(Object o);
public void setSecond(Object o);
public String toString ();
public boolean equals(Object obj);

}
```

Problem: Implement a Pair Of *Integers* in Java (3 designs)



Problem: Implement a Pair Of *Things* in Java

```
public abstract class PairAbstraction     implements
PairSpecification{

public abstract Object getFirst();

public abstract Object getSecond();

public abstract void setFirst(Object o);

public abstract void setSecond(Object o);
```

Problem: Implement a Pair Of *Things* in Java

```
publicvoid swap(){

        Object temp = getFirst();
        setFirst(getSecond());
        setSecond(temp);
}



public String toString (){

        return "("+getFirst()+","+getSecond()+")";
}
```

Problem: Implement a Pair Of *Things* in Java

```
 public boolean equals(Object obj){

       if (this == obj)     return true;
       if (obj == null)     return false;
       if (!(obj instanceof PairSpecification))
           return false;

       PairAbstraction other = (PairAbstraction) obj;

       return (other.getFirst() == getFirst() &&
               other.getSecond() == getSecond());

}

}// endclass PairAbstraction
```

Problem: Implement a Pair Of ***Integers*** in Java

```
public abstract class PairOfIntegersAbstraction
extends PairAbstraction implements PairSpecification{


public static final int DEFAULT_FIRST =0;
public static final int DEFAULT_SECOND =0;

public abstract void setFirst(Object o);

public abstract void setSecond(Object o);

public abstract Integer getFirst();

public abstract Integer getSecond();
```

Problem: Implement a Pair Of ***Integers*** in Java

```
public PairOfIntegersAbstraction(){
setFirst(DEFAULT_FIRST);
setSecond (DEFAULT_SECOND);
}

public PairOfIntegersAbstraction(PairOfIntegersAbstraction poi){
setFirst(poi.getFirst());
setSecond(poi.getSecond());
}

public PairOfIntegersAbstraction(Integer first, Integer second){
setFirst(first);
setSecond(second);
}

} // endclass PairOfIntegersAbstraction
```

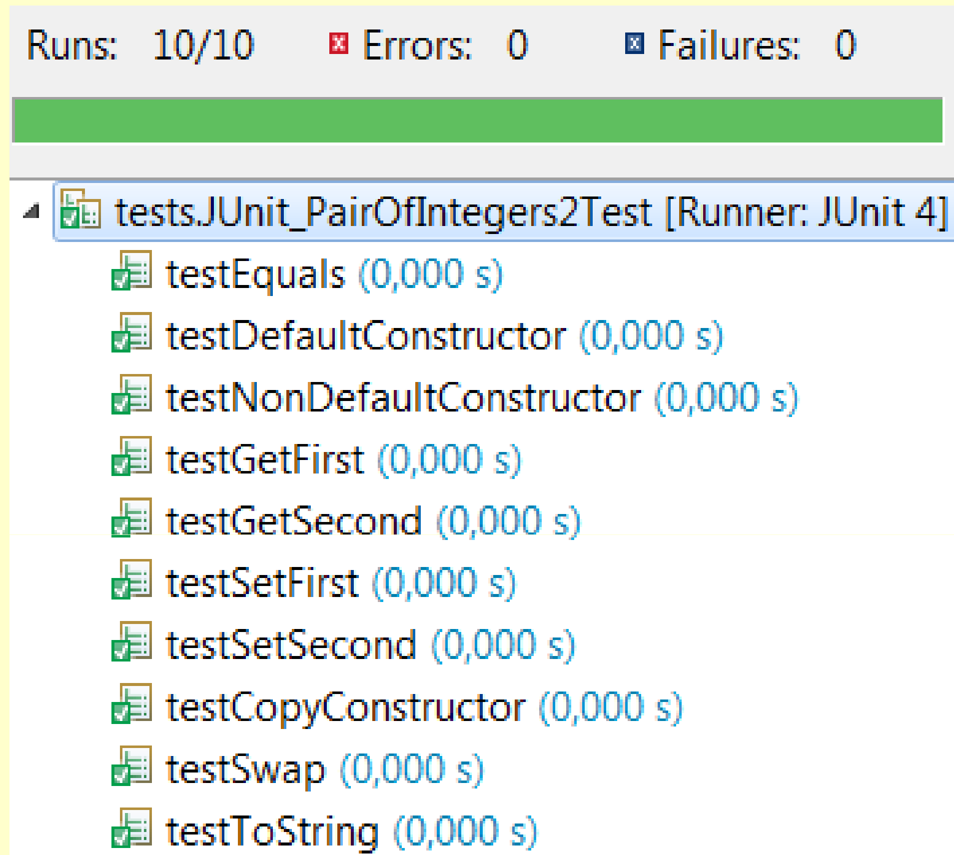
Problem: Implement a Pair Of ***Integers*** in Java (**2. without generics**)

```
public class PairOfIntegers2 extends PairOfIntegersAbstraction
implements HasInvariant, PairSpecification{

Integer first;
Integer second;

public PairOfIntegers2(){super();}

public PairOfIntegers2(PairOfIntegers2 poi){super(poi);}

public PairOfIntegers2(Integer first, Integer second){
super(first,second);
}

public Integer getFirst() {return first;}

public Integer getSecond() {return second;}
```

Problem: Implement a Pair Of ***Integers*** in Java (**2. without generics**)

```
public void setFirst(Object o) throws IllegalArgumentException{
if (!( o instanceof Integer))throw (new
IllegalArgumentException("Can't assign a non Integer"));
first = (Integer) o;
}


public void setSecond(Object o) throws IllegalArgumentException
{
if (!( o instanceof Integer)) throw (new
IllegalArgumentException("Can't assign a non Integer"));
second = (Integer) o;
}

} // endclass PairOfIntegers2
```

Problem: Implement a Pair Of ***Integers*** in Java (**2. without generics**)



TO DO: Run the Unit tests and check that the implementation is correct.

QUESTION: Do you understand how the Unit tests work, what they are testing and why they are structured as they are?

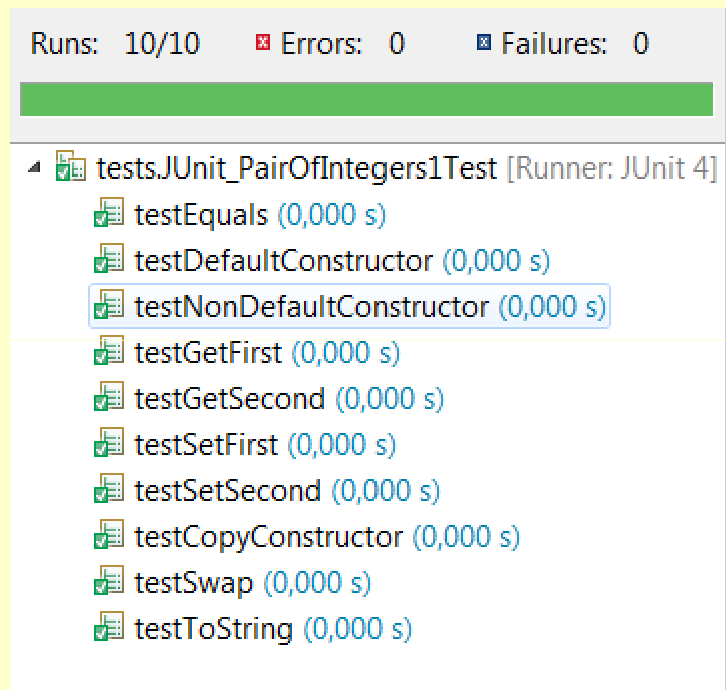
Problem: Implement a *Pair Of* ***Integers*** (**1. using user defined generics**)

We can also choose to implement the PairOfIntegers using the **generic pair** as follows:

```
public class PairOfIntegers1 extends PairOfIntegersAbstraction
implements  PairSpecification{

GenericPair <Integer> pairIntegers;

/* TO DO - complete the code here in order that the unit tests
   in JUnit_PairOfIntegers1Test pass.

   The Unit test code can be downloaded from the web site

} // endclass PairOfIntegers1
```

Problem: Implement a *Pair Of* ***Integers*** (**1. using user defined generics**)

TO DO: Implement and test a class **`PairOfIntegers1`**



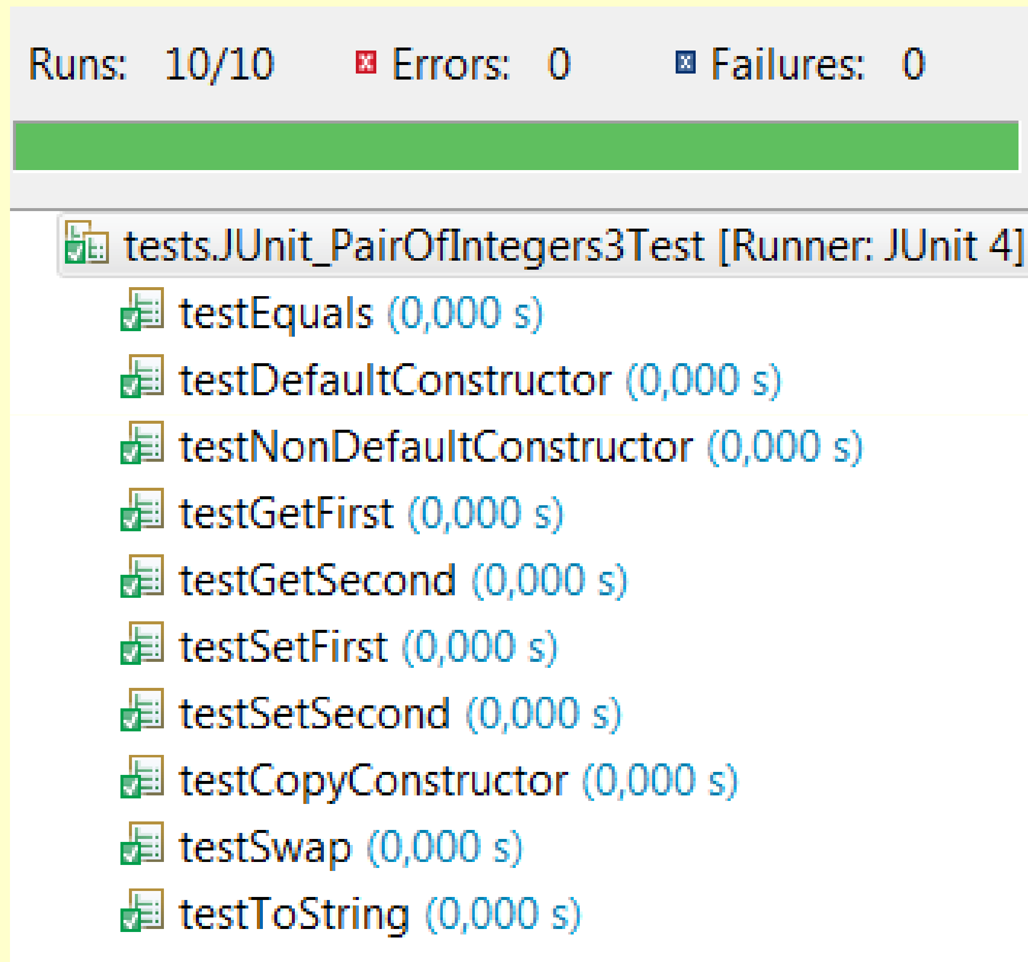
Problem: Implement a *Pair Of* ***Integers*** (3. using library generics)

We can also choose to implement the PairOfIntegers using the **generic libraries** as follows:

```
public class PairOfIntegers3 extends PairOfIntegersAbstraction
implements  PairSpecification{

ArrayList <Integer> pair;

/* TO DO - complete the code here in order that the unit tests
   in JUnit_PairOfIntegers3Test pass.

   The Unit test code can be downloaded from the web site

} // endclass PairOfIntegers3
```

Problem: Implement a *Pair Of* ***Integers*** (**1. using user defined generics**)

TO DO: Implement and test a class **`PairOfIntegers3`**



2. Exceptions – General History

The control of exceptional conditions in PL/I object programs
JM Noble - **Proc**. **IFIP Congress**, **1968.**

Exception Handling: Issues and a Proposed Notation
John B. Goodenough **Commun. ACM, 1975**

Software reliability: The role of programmed exception handling, **Melliar-Smith, P. M. and Randell, B, SIGSOFT Softw. Eng. Notes, 1977**

Exception Handling in CLU, **Liskov, B.H.; Snyder, A.**; **Software Engineering, IEEE Transactions, 1979**

A modular verifiable exception handling mechanism , **Shaula Yemini and Daniel M. Berry. 1985.. ACM Trans. Program. Lang. Syst**

2. Exceptions – Further Reading (C++ and Java)

Exception Handling for C++, **A. R. Koenig and B. Stroustrup: Journal of Object Oriented Programming, 1990**

Analyzing exception flow in Java programs. **Martin P. Robillard and Gail C. Murphy. In Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering. 1999**

Analysis and testing of programs with exception handling constructs , **Sinha, S.; Harrold, M.J.; Software Engineering, IEEE Transactions Sep 2000**

A comparative study of exception handling mechanisms for building dependable object-oriented software, **Alessandro F. Garcia, Cecilia M. F. Rubira, Alexander Romanovsky, Jie Xu, Journal of Systems and Software, Volume 59, Issue 2, 15 November 2001**

Exceptions in Java

When a method encounters an abnormal condition (an *exception condition*) that it can't handle itself, it may *throw* an exception.

Exceptions are *caught* by handlers positioned along the thread's method invocation stack. If the calling method isn't prepared to catch the exception, it throws the exception up to *its* calling method, and so on.

If one of the threads of your program throws an exception that isn't caught by any method along the method invocation stack, that thread will expire.

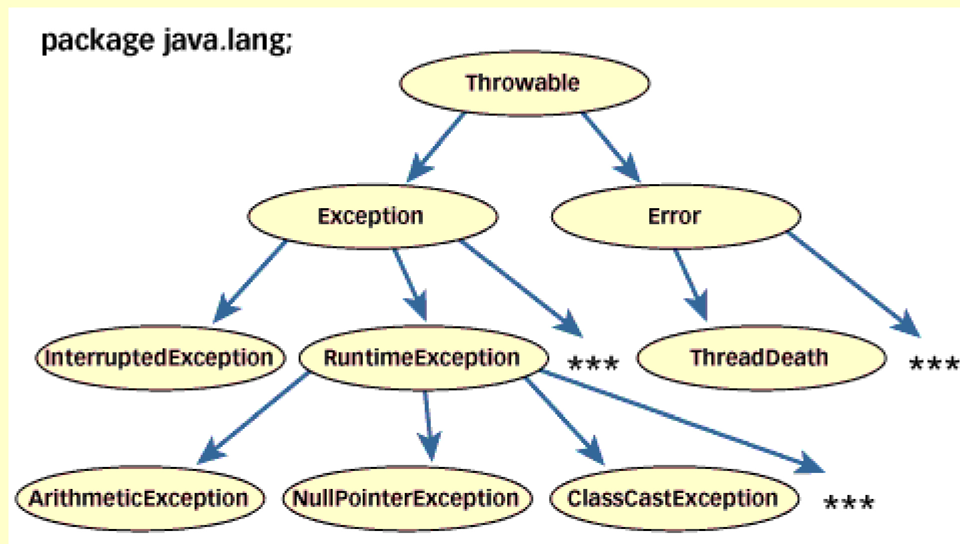
When you program in Java, you must position catchers (the exception handlers) strategically, so your program will catch and handle all exceptions from which you want your program to recover.

Exceptions in Java

In Java, exceptions are objects. When you throw an exception, you throw an object.

You can't throw just any object as an exception, however -- only those objects whose classes descend from Throwable.

Throwable serves as the base class for an entire family of classes, declared in java.lang, that your program can instantiate and throw.



Exceptions in Java

Exceptions (members of the Exception family) are thrown to signal abnormal conditions that can often be handled by some catcher, though it's possible they may not be caught and therefore could result in a dead thread.

Errors (members of the Error family) are usually thrown for more serious problems, such as OutOfMemoryError, that may not be so easy to handle.

In general, code you write should throw only exceptions, not errors. Errors are usually thrown by the methods of the Java API, or by the Java virtual machine itself.

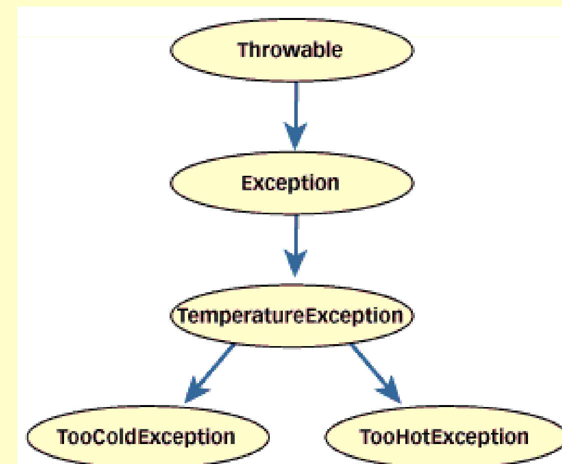
In addition to throwing objects whose classes are declared in java.lang, you can throw objects of your own design. To create your own class of throwable objects, you need only declare it as a subclass of some member of the Throwable family. In general, however, the throwable classes you define should extend class Exception.

Exceptions in Java

Whether you use an existing exception class from java.lang or create one of your own depends upon the situation. In some cases, a class from java.lang will do just fine.

For example, if one of your methods is invoked with an invalid argument, you could throw IllegalArgumentException, a subclass of RuntimeException in java.lang.

Sometimes you will want to indicate that a method encountered an abnormal condition that isn't represented by a class in the Throwable family of java.lang.
For example, in a coffee machine:



NOTE: Exceptional conditions are not necessarily rare, just outside the normal flow of events.

Exceptions in Java: Example Coffee Cup

```
public void drinkCoffee(CoffeeCup cup) throws TooColdException,
TooHotException {

 int temperature = cup.getTemperature();

 if (temperature <= TOOCOLD) throw new TooColdException();
 else if (temperature >= TOOHOT) throw new TooHotException();
 else cup.sip();

}
```

```
try {
cust.drinkCoffee(cup); System.out.println("Coffee is just right.");
}
catch (TooColdException e) { System.out.println("Coffee is too cold."); }
catch (TooHotException e) { System.out.println("Coffee is too hot."); }
 }
```

Exceptions in Java: Example Coffee Cup

You can also group catches:

```
try {
cust.drinkCoffee(cup);
System.out.println("Coffee is just right.");
}
catch (TemperatureException e) {
System.out.println("Coffee is too cold or too hot.");
}
```

QUESTION: What about throwing exceptions inside the catch?
Does Java allow this? What are the semantics/rules?

TO DO: Write some experimental code to find the answers to these questions.

Exceptions in Java: Embedding information in an exception object

When you throw an exception, you are performing a kind of structured go-to from the place in your program where an abnormal condition was detected to a place where it can be handled.

The Java virtual machine uses the class of the exception object you throw to decide which catch clause, if any, should be allowed to handle the exception.

But an exception doesn't just transfer control from one part of your program to another, it also transmits information. Because the exception is a full-fledged object that you can define yourself, you can embed information about the abnormal condition in the object before you throw it. The catch clause can then get the information by querying the exception object directly.

Exceptions in Java: Example Coffee Cup

```
class UnusualTasteException extends Exception {
UnusualTasteException() { }
UnusualTasteException(String msg) { super(msg);}
}

new UnusualTasteException("This coffee tastes like tea.")


try {
      cust.drinkCoffee(cup);
      System.out.println("Coffee ok.");
}
catch (UnusualTasteException e) {
System.out.println( "Customer is complaining of unusual taste.");
String s = e.getMessage();
if (s != null) System.out.println(s);
}
```

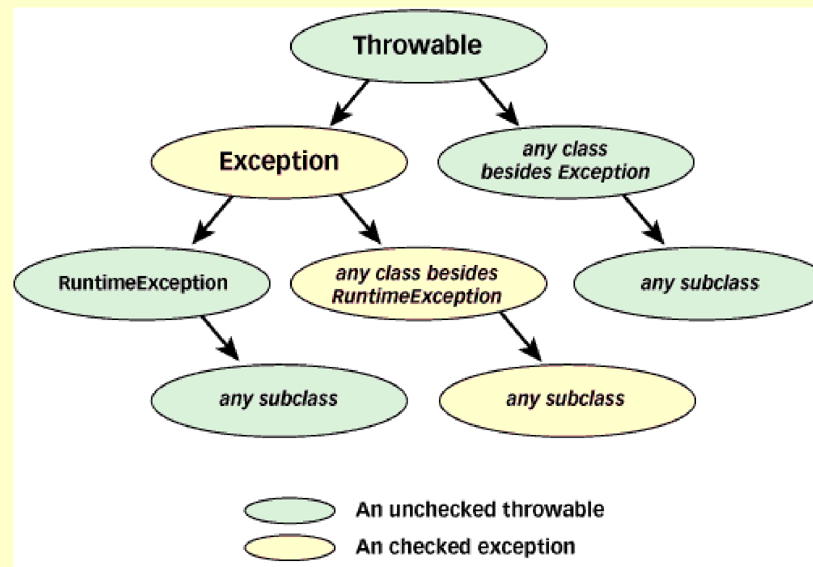
NOTE: here the info passed is a String explaining the strange taste, for TOOHOT or TOOCOLD we could pass the temperature value

Exceptions in Java: Checked vs. unchecked exceptions

There are two kinds of exceptions in Java, *checked* and *unchecked*, and only checked exceptions need appear in throws clauses.

The general rule is: Any checked exceptions that may be thrown in a method must either be caught or declared in the method's throws clause.

Checked exceptions are so called because both the Java compiler and the Java virtual machine check to make sure this rule is obeyed.



Exceptions in Java: finally block

Once a Java virtual machine has begun to execute a block of code -- the statements between two matching curly braces -- it can exit that block in any of several ways.

It could, for example, simply execute past the closing curly brace. It could encounter a break, continue, or return statement that causes it to jump out of the block from somewhere in the middle. Or, if an exception is thrown that isn't caught inside the block, it could exit the block while searching for a catch clause.

Given that a block can be exited in many ways, it is important to be able to ensure that something happens upon exiting a block, no matter how the block is exited. For example, if you open a file in a method, you may want to ensure the file gets closed no matter how the method completes. In Java, you express such a desire with a finally clause.

```
try { // Block of code with multiple exit points }
 finally {
 /* Block of code that must always be executed when the try
block exited, no matter how the try block is exited */
}
```

Java: Overriding methods that throw exceptions

TO DO: Write some experimental code to see if you can identify the rules for overiding methods that throw exceptions.

QUESTION: Can the new method (in the subclass)

- Add a new exception?
- Specialise/Generalise an exception thrown by the base class
- Ignore an exception thrown by the base class?

Processes and Threads

•The two fundamental units of execution in a concurrent program.

•In Java, concurrent programming is mostly thread-based.

•Processing time for each core in a system is shared among processes and threads through an OS feature called time slicing.

•Concurrency is possible even on simple systems, without multiple processors or execution cores.

Processes

Self-contained execution environment.

Independent set of basic run-time resources, such as memory space.

A single application may be implemented by a set of cooperating processes.

Most operating systems support *Inter Process Communication* (IPC) resources.

IPC can also used for communication between processes on different systems.

Most implementations of the JVM run as a single process.

Threads

Also known as *lightweight processes*.

Creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one.

Threads share the process's resources, including memory and open files.

This has advantages and disadvantages … can you think of them?

Multithreaded execution is essential in Java:

- every application has at least one thread
- "system" threads that do memory management, event/signal handling, etc.

In programming, we start with just one thread, called the *main thread*.

Any thread (including the main thread) can create new threads.

Thread Example

```
public class ThreadExample {


    public static void main (String[] args) {
        System.out.println("Starting Thread main");
        new SimpleThread("Add1", '1').start();
        new SimpleThread("Add2", '2').start();
        System.out.println("Finishing Thread main");
    }
}
```

Thread Example - typical output

```
Starting Thread main
Finishing Thread main
String Add2 extended to 2
String Add2 extended to 22
String Add2 extended to 222
String Add1 extended to 1
String Add1 extended to 11
String Add2 extended to 2222
String Add2 extended to 22222
No more increments left for threadAdd2
String Add1 extended to 111
String Add1 extended to 1111
String Add1 extended to 11111
No more increments left for threadAdd1
```

Thread Example - SimpleThread Code

```
class SimpleThread extends Thread {

// see http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html

String stringofchars;
char increment;

    public SimpleThread(String str,  char inc) {
        super(str);
        stringofchars = "";
        increment = inc;
    }
    public void run() {
        for (int i = 0; i < 5; i++) {

            try {
                sleep((int)(Math.random() * 3000));
            } catch (InterruptedException e) {}
        stringofchars = stringofchars + increment;
        System.out.println("String " + getName()+
                                  " extended to "+ stringofchars );
        }
        System.out.println("No more increments left for
                                     thread" + getName());
    }
}
```

IllegalThreadStateException

The runtime system throws an `IllegalThreadStateException` when you call a method on a thread and that thread's state does not allow for that method call. (See the state machine diagram in later slides)

So, when you call a thread method that can throw an exception, you must either catch and handle the exception, or specify that the calling method throws the uncaught exception.

The sleep method can also throw an `InterruptedException`, and so we needed a try/catch in the previous code:

```
try {
        sleep((int)(Math.random() * 3000));
    } catch (InterruptedException e) { }
```

Sharing Thread Problem

The previous example showed how two independent threads execute concurrently.

Threads can also share data/objects and so their concurrent behaviours are inter-dependent.

We wish to change the previous code so that the 2 threads update the same string of characters.

We will do this using a SharedString class

Sharing Thread Problem

```java
 class SharedString {

public SharedString(){str ="";}

public String str;

public  void add (char c){str = str + c;}

public String toString () {return str;}
}


public class SharingThreads {

    public static void main (String[] args) {

    SharedString soc = new SharedString();
        new SharingThread("SharingAdda", soc, 'a').start();
        new SharingThread("SharingAddb", soc, 'b').start();
    }

}
```

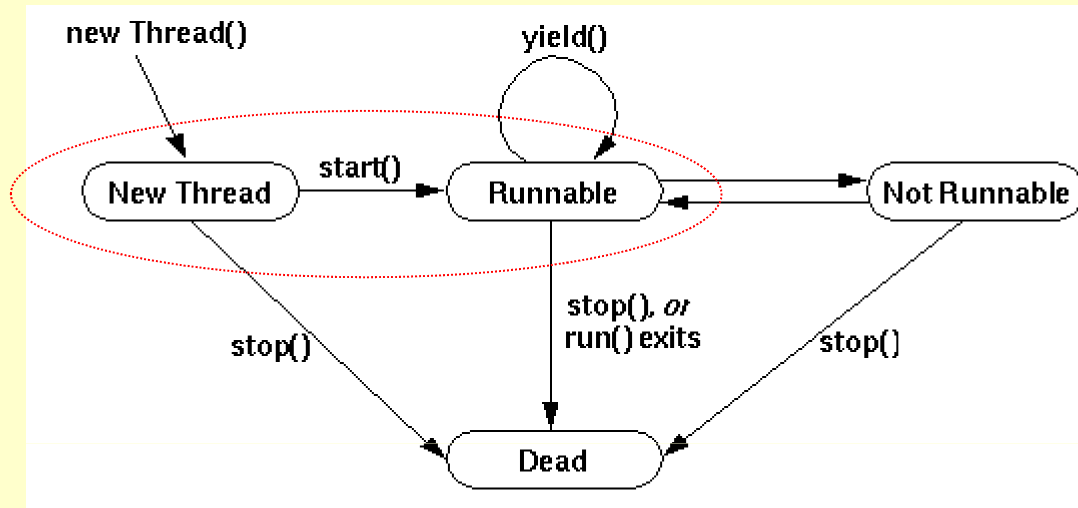
Sharing Thread Problem

We want the output from this code to produce, typically:

```
Shared String extended by SharingAddb to b
Shared String extended by SharingAddb to bb
Shared String extended by SharingAdda to bba
Shared String extended by SharingAddb to bbab
Shared String extended by SharingAddb to bbabb
Shared String extended by SharingAdda to bbabba
Shared String extended by SharingAddb to bbabbab
No more increments left SharingAddb
Shared String extended by SharingAdda to bbabbaba
Shared String extended by SharingAdda to bbabbabaa
Shared String extended by SharingAdda to bbabbabaaa
No more increments left SharingAdda
```

TO DO: Your task is to code the **`class`** `SharingThread` **`extends`** `Thread {}` to provide this behaviour

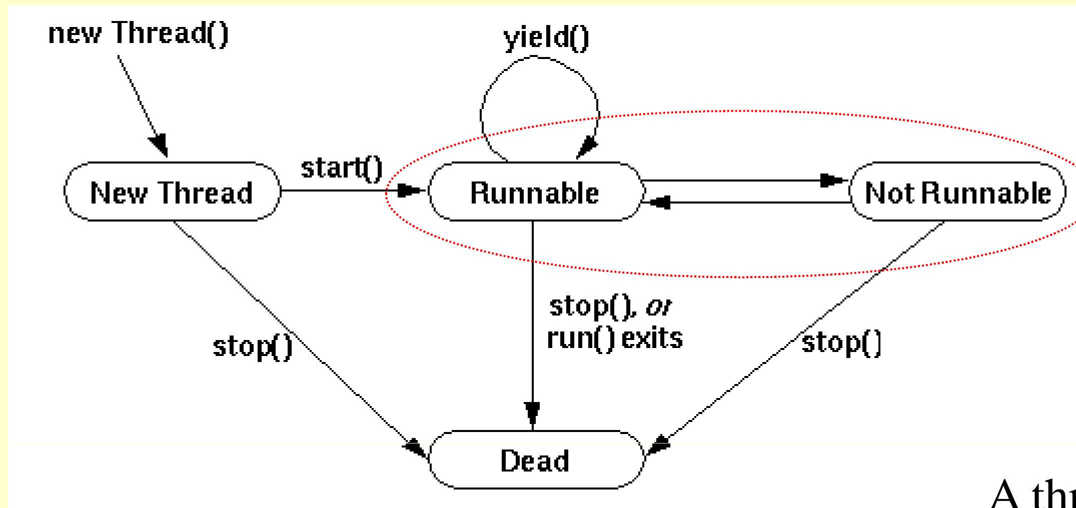
Thread State Machine: an abstraction of the complete diagram



The `start()` method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's `run()` method.

The next state state is "`Runnable`" rather than "`Running`" because the thread might not actually be running when it is in this state.

Thread State Machine: an abstraction of the complete diagram



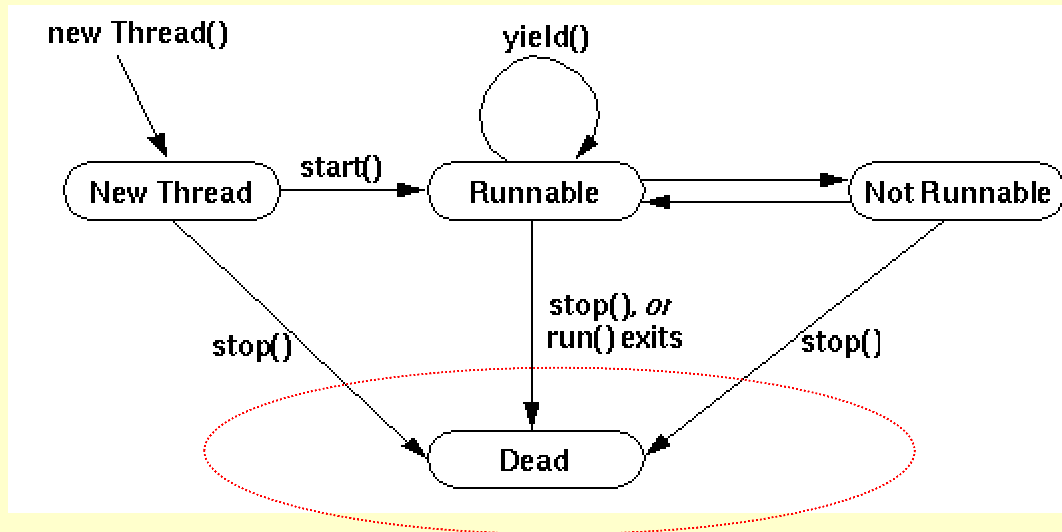
A thread enters the "`Not Runnable`" state when:

- `sleep()` is called.
- `suspend()` is called.
- The thread uses its `wait()` method to wait on a condition variable.
- The thread is blocking on I/O.

A thread leaves the "`Not Runnable`" state when a matching condition is met:

- `sleep()` is completed.
- `resume()` is called
- object owning the variable calls `notify()` or `notifyAll()`
- I/O completes

Thread State Machine: an abstraction of the complete diagram



A thread dies naturally when its `run()` method exits normally

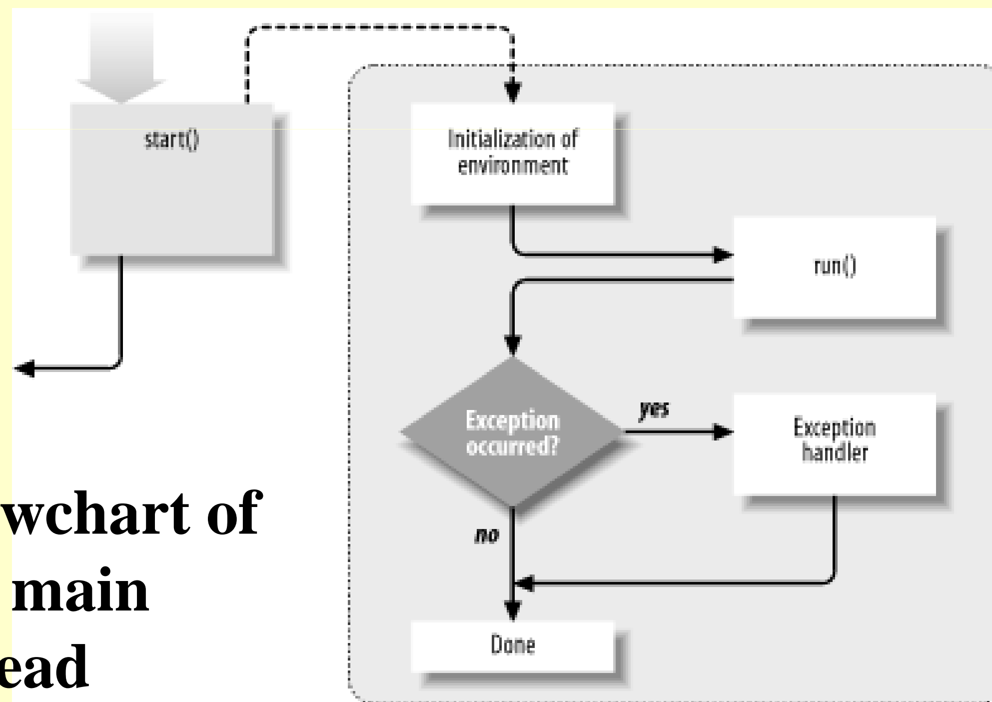
You can also kill a thread at any time by calling its `stop()` method

QUESTION: What should happen if an exception occurs inside a thread?

Threads and exceptions: an aside

The start() method does start another thread of control, but the run() method is not really the "main" method of the new thread.

The run() method is executed inside a context that allows the virtual machine to handle runtime exceptions thrown from the run() method.



Flowchart of the main thread

Threads and exceptions: an aside

All uncaught exceptions are handled by code outside of the run() method before the thread terminates. The default exception handler is a Java method; it can be overridden. This means that it is possible for a program to write a new default exception handler.

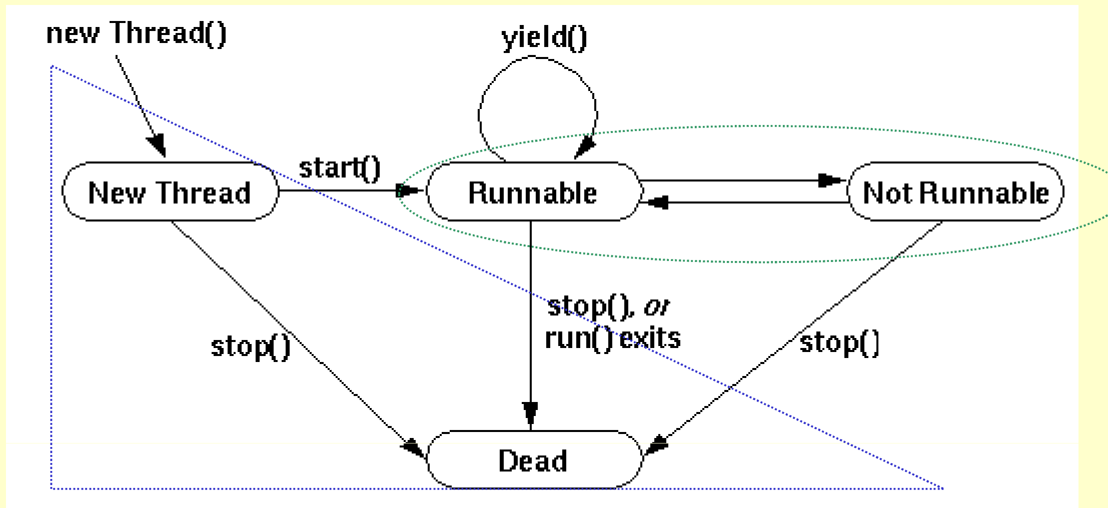
The default exception handler is the uncaughtException() method of the ThreadGroup class. It is called only when an exception is thrown from the run() method. The thread is technically completed when the run() method returns, even though the exception handler is still running the thread.

The default implementation of the uncaughtException() method is to print out the stack trace of the Throwable object thrown by the run() method

In most cases, this is sufficient: the only exceptions that the run() method can throw are runtime exceptions or errors. By the time the run() method has returned, it's too late to recover from these errors.

One case in which it's useful to override the uncaughtException() method is to send a priority notification to an administrator that an unusual, fatal error has occurred. Here's an example that does that when its thread eventually encounters an out-of-memory error:

Thread State Machine: an abstraction of the complete diagram



The `isAlive()` method returns true if the thread has been started and not stopped.

`thread.isAlive() = false` => `thread` is either a "`New Thread`" or "`Dead`".

`thread.isAlive() = true` => `thread` is either a "`Runnable`" or "`Not Runnable`".

Threads and Synchronization Issues

Threads can share state (objects)

This is very powerful, and makes for very efficient inter-thread communication

However, it makes two kinds of errors possible:

- *thread interference*, and
- *memory inconsistency*.

Java provides a *synchronization "tool"* in order to avoid these types of errors.

Thread Interference

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

Consider a simple class called Counter

```
class Counter {
    private int c = 0;
            public void increment() {c++;}

    /*              Multiple steps of c++
             1.  Retrieve the current value of c.
            2.  Increment the retrieved value by 1.
           3.  Store the incremented value back in c.
    */

    public void decrement() {c--;}
    public int value() {return c;}
}
```

If a Counter object is referenced from multiple threads, interference between threads may give rise to unexpected behaviour.

Memory inconsistency

Consider the following example.

```
int counter = 0;
```

The counter field is shared between two threads, A and B.

Suppose thread A increments counter:

```
counter++;
```

Then, shortly afterwards, thread B prints out counter:

```
System.out.println(counter);
```

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be `"1"`.

But, in this example, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a ***happens-before*** relationship between these two statements.

There are several actions that create ***happens-before*** relationships.

The simplest technique/tool is to use ***`synchronization`***

Synchronized methods, example:

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment()
{c++;}
    public synchronized void decrement() {c-
-;}
    public synchronized int value() {return
c;}
}
```

Two invocations of synchronized methods on the same object cannot interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

When a synchronized method exits, it automatically establishes a *happens-before* relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Constructors cannot be synchronized

Sychronization is effective for keeping systems *safe*, but can present problems with *liveness*

3. Reflection

Reflection is the process by which a program can read its own **metadata** (data about data).

A program is said to reflect on itself, extracting metadata from its assembly and using that metadata either to inform the user or to modify its own behavior.

In an object-oriented world, metadata is organized into objects, called **metaobjects**. The runtime self-examination of the metaobjects is called **introspection**.

Reflection is important since it lets you write programs that does not have to "know" everything at compile time, making them more **dynamic**, since they can be tied together at **runtime**.

Applications programmed (cleanly) with reflection **adapt** more easily to **changing requirements**. Well programmed reflective components are more likely to be **reused** flawlessly in other applications.

Reflection: Some Background/Further Reading

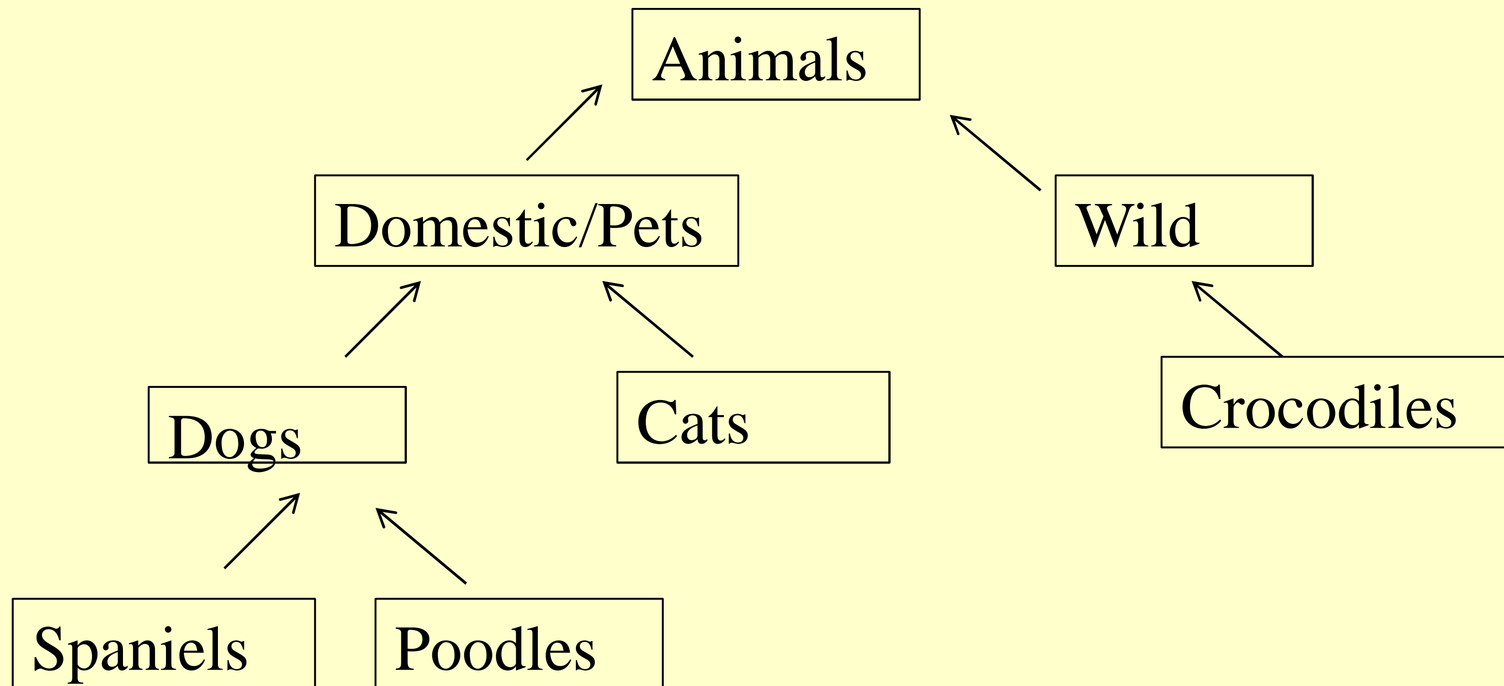
Reflection and semantics in LISP. **Brian Cantwell Smith. Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '84). ACM**

Reflection in logic, functional and object-oriented programming: a Short Comparative Study, **Francois-Nicola Demers and Jacques Malenfant, Proc. of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI. pp. 29–38. August 1995**

Java Reflection in Action (In Action Series). **Ira R. Forman and Nate Forman. 2004 Manning Publications Co., Greenwich, CT, USA.**

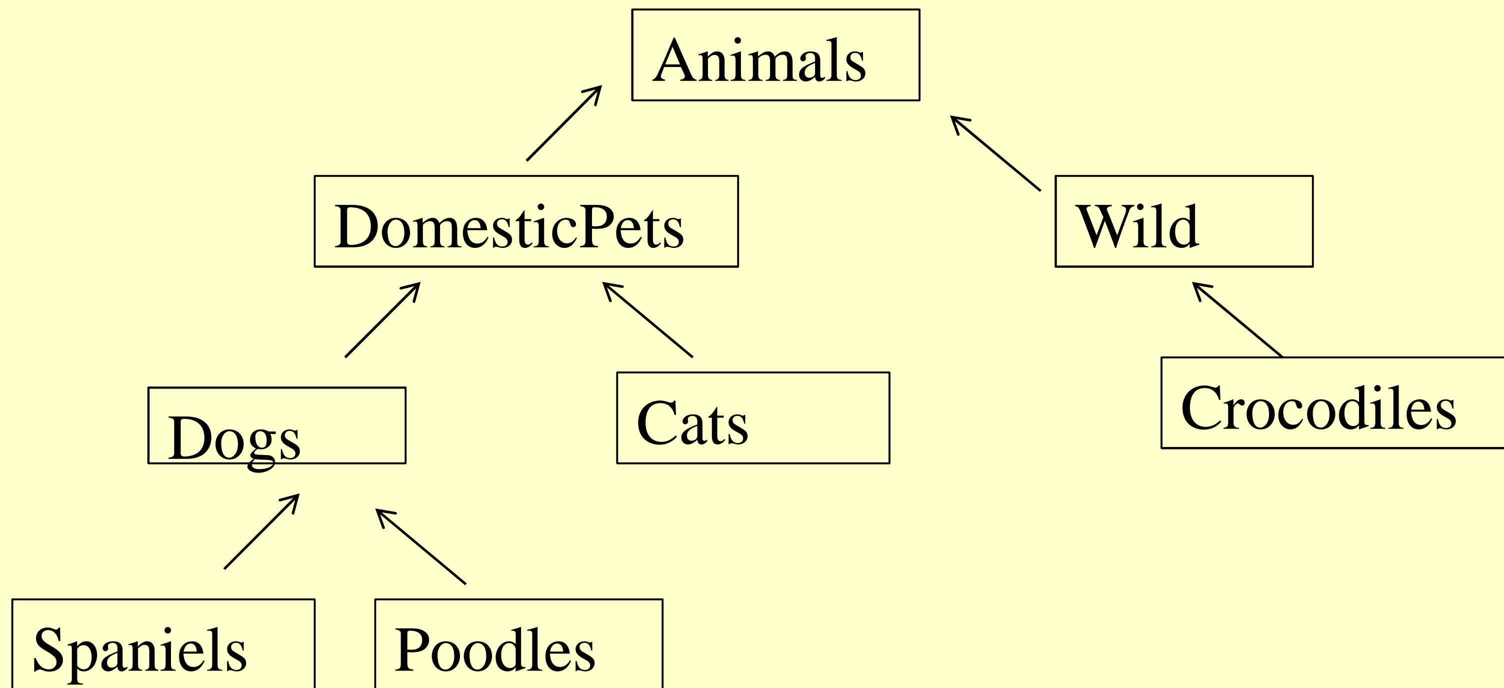
Reflection PBL – LeastAbstractCommonSuperclass (LACS)

Consider the following class hierarchy



Imagine if we had a generic collection of Animals and that we wished to examine all elements of the collection and find the least abstract subclass to which all these Animals belonged

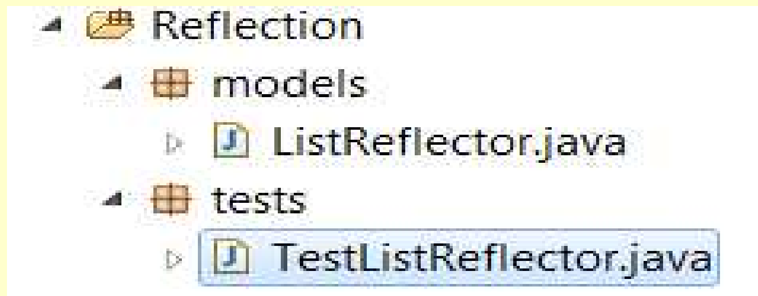
Reflection PBL - LeastAbstractCommonSuperclass



LACS {spaniel, poodle, poddle, cat, spaniel} = DomesticPets
LACS {spaniel, poodle, poddle, dog, spaniel} = Dogs
LACS {spaniel, poodle, crocodile} = Animals

TO DO: Test the developed code on these 3 example cases

Reflection PBL - LeastAbstractCommonSuperclass



TO DO: Download the Reflection outline code from the website and try to understand what it is doing

The `ListReflector` class provides to methods that require the use of reflection:

void models.ListReflector.reflect(List<T> list)

Parameters:

<**T**> is the generic type of list elements

list is the list elements whose information (gathered using reflection) will be printed to the screen

Class<? extends Object> models.ListReflector.lowestCommonSuperclass(List<T> listOfObjects)

Parameters:

<**T**> is the

listOfObjects

Returns:

the most concrete class of which all the list elements are members

@todo

The students are to write this code so that it functions correctly as tested in `TestListReflector`.

Reflection PBL - LeastAbstractCommonSuperclass

The `TestListReflector` should test the that the LACS requirement is correctly fulfilled

Author:

J Paul Gibson Template test code for reflection problem (OOD)

@todo

The students are to improve the test to show that the method `ListReflector.lowestCommonSuperclass` is currently not working correctly

They are then to fix the method `ListReflector.lowestCommonSuperclass` and show that their fix is correct (by executing the updated test)

Objective: You will learn about reflection from trying to solve this problem