Announcements

- Homework 3 deadline – Mon, 10/30, 11:55PM
- Midterm – Wed, 11/01
  - Same room as lecture
  - Same time
  - Open books, open notes

Midterm

- No electronic devices
- Study class notes
- Practice homework problems (1,2,3) again
- Everything covered until Monday, 10/30
- Sample midterm under CCLE
- Bring everything
  - printouts of homework, solutions
  - Class notes, section notes
  - Textbook

What's happening today?

- Java
  - Static, final keywords
  - Polymorphism, dynamic dispatch
  - Exceptions
- Sample of midterm related questions

Static, Final keywords

Reminder: Static keyword

- The keyword *static* determines whether the variable is a *class variable* or an *instance variable*
- To access a static variable you must qualify it with the class name (NOT the instance name)
  - for example:
  - `ClassName.theVariableName = aValue;`
- Why should we use static variable?

Final keyword

- Determines whether is a constant or not
- Example:
  - declaring a final object and trying to assign a new value to it
- There’s a difference between declaring final and immutability
- ‘final’ means you can’t change the reference, that’s all

Final keyword

- Final fields
  - Marking fields as final forces them to be set by end of construction
  - Immutable, must be declared before constructor finishes: `class x { final int myField = 5; }`
- Final static fields
  - In general, static means "associated with the *type* itself, rather than an *instance* of the type."
  - Like constants: `private final static int NUMBER = 10;`
- Final classes
  - A class that can't be extended.
- Final methods
  - Basically same as final classes, cannot be changed by subclasses

Final keyword

```
abstract class Animal {
    abstract int legs();
}

class Human extends Animal {
    @Override
    final int legs() { return 2; } // Add a final here
}

class Student extends Human {
    int getAverage() { return 1; }
    @Override
    int legs() { return 10; } // We don't want that!!
}
```

Q: What happens if you make an array “final”? Can you change its size? Can you change the values inside? Try it out!

Polymorphism and Dynamic Dispatch

Dynamic dispatch

- Dynamic - associated with runtime
- Static - associated with compile-time
- The default in Java is Dynamic
  - Methods are bound *dynamically* according to the type of object referenced at run-time
  - Unless the method is declared final, then the binding is *static*
    - More on this later...

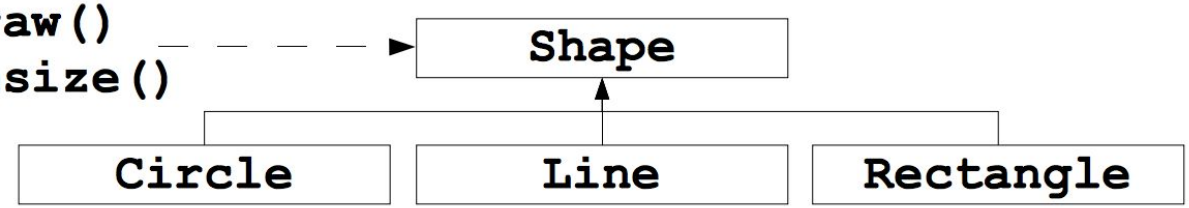
Polymorphism and Dynamic Dispatch

```
class A {
  void callme() {
    System.out.println("Inside A's callme method");
  }
}

class B extends A {
  void callme() {
    System.out.println("Inside B's callme method");
  }
}

class C extends A {
  void callme() {
    System.out.println("Inside C's callme method");
  }
}
```

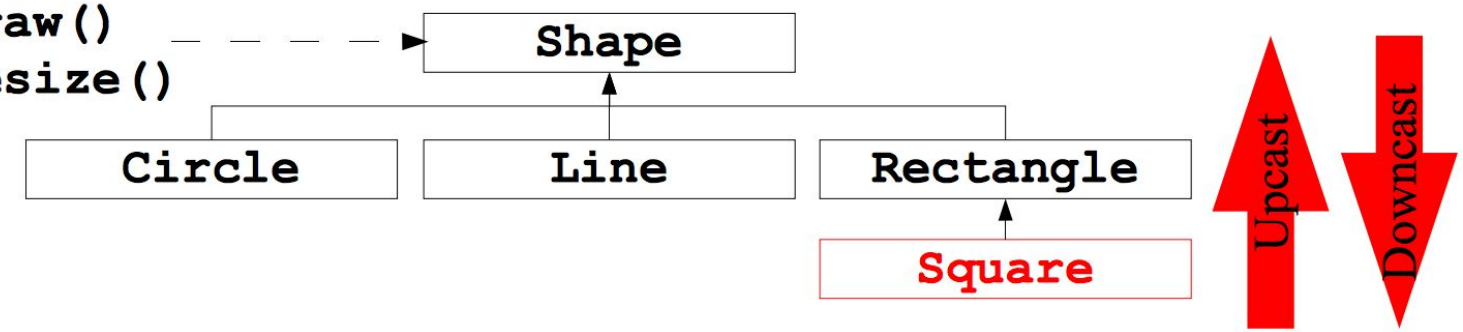
```
class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A

    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme

    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
```

Polymorphism Goals

```
// abstraction (and substitutability)
Shape s;
s.draw()
s.resize()
```




```
// extensibility
Shape s;
s.draw()
s.resize()
```



Polymorphism

- Polymorphism: “The ability of a variable or argument to refer at run-time instances of various classes”.

```
Shape s = new Shape();
Circle c = new Circle();
Line l = new Line();
Rectangle r = new Rectangle();

s = l;          // is this legal?
l = s;          // is this legal?
l  = (Line)s    // is this legal?
```

- The assignment **`s=l`** is legal if the static type of **`l`** is **`Shape`** or subclass of **`Shape`**
- This is static type checking where the type comparison rules can be done at compile-time
- Polymorphism is constrained by the inheritance hierarchy

Static and Dynamic Type

- The static type of a variable/argument is the declaration type
- The dynamic type of a variable/argument is the type of the object the variable/argument refers to.

```
class A{
  // body
}
class B extends A{
  // body
}

public static void main(String args[]){
   A x;              // x static type A
   B y;              // y static type B

   x = new A();      // x dynamic type A
   y = new B();      // y dynamic type B
   x = y;            // x dynamic type B
}
```

Method Binding

```
class A {
   void doSomething(){
      ...
   }
}
```

```
class B extends A {
   void doSomething (){
      ...
   }
}
```

```
A x = new A();
B y = new B();
x = y;
x.doSomething(); // on class A or class B?
```

- *Binding*: Connecting a method call to a method body.
- *Dynamic binding*: The dynamic type of **x** determines which method is called - not possible without polymorphism.
- *Static binding*: The static type of **x** determines which method is called

Dynamic Binding Example

```
public class Shape {
   void draw() { System.out.println ("Shape"); }
}
public class Circle extends Shape {
   void draw() { System.out.println ("Circle"); }
}
public class Line extends Shape {
   void draw() { System.out.println ("Line"); }
}
public class Rectangle extends Shape {
   void draw() {System.out.println ("Rectangle"); }
}

public static void main(String args[]){
   Shape[] s = new Shape[3];
   s[0] = new Circle();
   s[1] = new Line();
   s[2] = new Rectangle();
   for (int i = 0; i < s.length; i++){
      s[i].draw(); // prints Circle, Line, Rectangle
   }
}
```

Dynamic Binding and Constructors

```
public class A { // example from inheritance lecture
   public A(){
      System.out.println("A()");
      // when called from B the B.doStuff() is called
      doStuff();
   }
   public void doStuff(){ System.out.println("A.doStuff()"); }
}

public class B extends A{
   int i = 7;
   public B(){ System.out.println("B()"); }
   public void doStuff(){System.out.println("B.doStuff() " + i);}
}

public class Base{
   public static void main(String[] args){
      B b = new B();
      b.doStuff();
   }
}
```

```
//prints
A()
B.doStuff() 0
B()
B.doStuff() 7
```

Dynamic Binding and *private* Methods

```
class Shape {
   void draw() { System.out.println ("Shape"); }
   private void doStuff() {
      System.out.println("Shape.doStuff()");
   }
}
class Rectangle extends Shape {
   void draw() {System.out.println ("Rectangle"); }
   public void doStuff() {
      System.out.println("Rectangle.doStuff()");
    }
}

public class PolymorphShape {
   public static void polymorphismPrivate(){
      Rectangle r = new Rectangle();
      r.doStuff();   // okay part of Rectangle interface
      Shape s = r;   // upcast
      s.doStuff();   // not allowed, compiler error
   }
}
```

Why Polymorphism and Dynamic Binding

- Separate interface from implementation
  - Encapsulation
  - We are trying to achieve Object-Oriented programming!
- Allows programmers to isolate type specific details from the main part of the code
  - E.g. Client programs only use the method provided by the `Shape` class in the shape hierarchy example
- Code is simpler to write and read
  - Abstraction
- Can change types (add new types) without the changes propagating to existing code

Polymorphism Summary

- Polymorphism is an object-oriented “switch” statement
- Polymorphism is a prerequisite for dynamic binding and central to the object-oriented programming paradigm
- Abstract classes

Questions - what is the output?

```
class Point {
     public int x, y;
     public boolean isEqual(Point p) {
          return p.x == x && p.y == y;
     }
}

class ColoredPoint extends Point {
     public int color;
     public Boolean
isEqual(ColoredPoint p) {
          return p.x == x && p.y == y
&& p.color == color;
     }
}
```

```
void f () {
     ColoredPoint p,q;
     p = new ColoredPoint();
     p.x = 1; p.y = 2; p.color = 3;
     q = new ColoredPoint();
     q.x = 1; q.y = 2; q.color = 4;
     boolean b1 = p.isEqual(q); // b1???
     Point r = (Point) q;
     boolean b2 =  p.isEqual(r); // b2???
}
```

Questions - what is the output?

```
class OurInt {
int i;
public OurInt(int i) {
     this.i = i;
}
public void set(int j) {
     i = j;
}
public int get() {
     return i;
     }
}
```

```
public class SimpleTest {
     static void increment(OurInt i) {
          i.set(20);
          i = new OurInt(33);
     }
     public static void main(String[] args) {
          OurInt i = new OurInt(10);
          increment(i);
          System.out.println(i.get());
          //What will i be here?
     }
}
```

Questions - what is the problem?

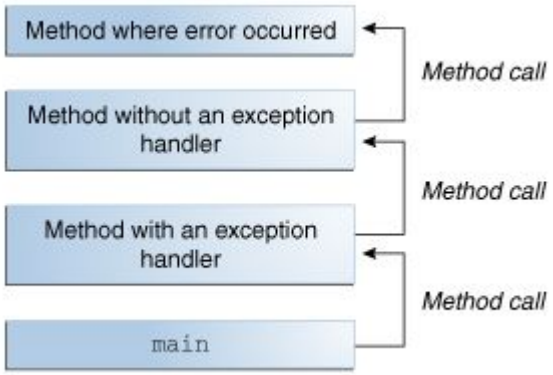
What is the problem here?

```
class B {
    protected int get() {...}
}
class A extends B {
    private int get() {...}
}
class C extends B {
    public int get() {...}
}
```

Exceptions

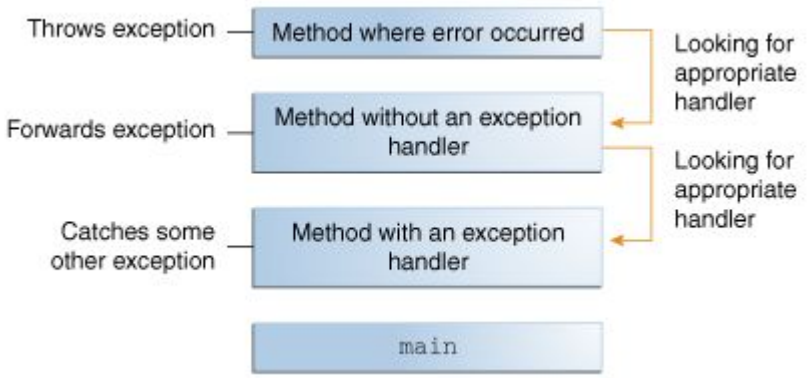
Exceptions

- Sometimes your program does something it shouldn’t do, like divide by zero, or read from a non-existing file
- When such event happens, an exception object is created
- Creating an exception and handing it to the system is called *throwing an exception*
- Runtime attempt to find something to handle the exception



Exceptions

- Sometimes your program does something it shouldn't do, like divide by zero, or read from a non-existing file
- When such event happens, an exception object is created
- Creating an exception and handing it to the system is called *throwing an exception*
- Runtime attempt to find something to handle the exception



Control flow and Exceptions

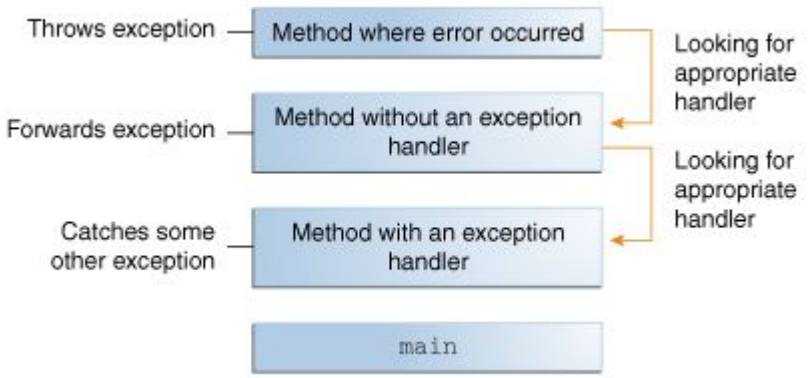
- When exception is thrown control returns through the methods called in reverse calling order until a try statement is found with a catch block for the exception

- It is possible for a catch statement to defer handling of an exception by including a throw statement of its own

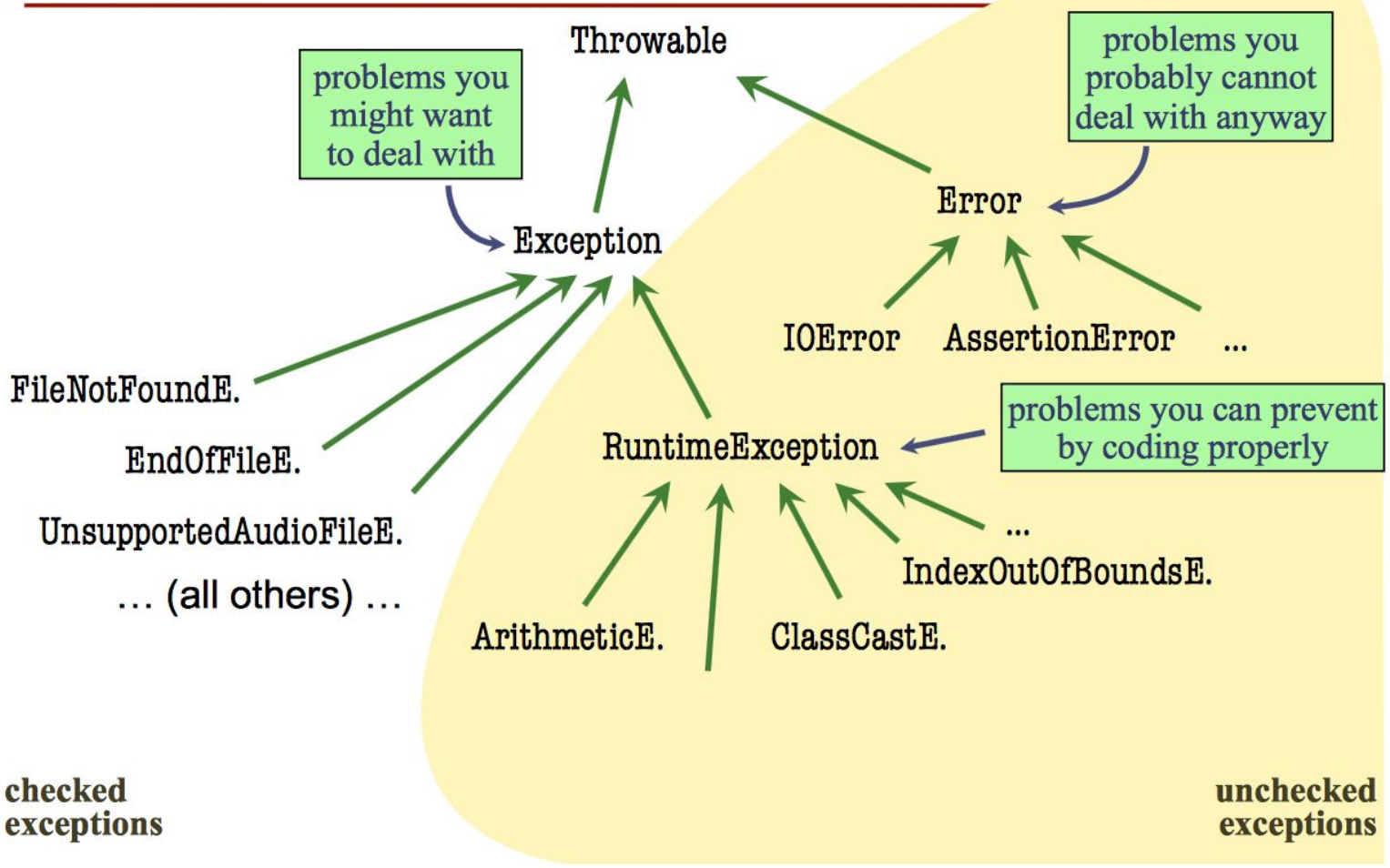
Catching Exceptions

- The exception handler chosen is said to catch the exception
- Catch exception with try block:

```
try{
 // some code...
}
catch(BadException e)
{ //executed if exception is of type BadException
//code to recover from error...
}
catch(AnotherException e)
{ //executed if exception is of type AnotherException
//code to recover from error...
}
Finally
{ //executed always
//code...
}
```



Exception Hierarchy



throws and checked Exceptions

- throws is needed if
  - The method itself throws a checked exception
  - The method calls a method that throws a checked exception
- throws is not needed if
  - All checked exceptions are caught
  - Any uncaught exceptions are unchecked exceptions

throws and checked Exceptions

```
public class Ex {
   public static void first() {
      try {
         second();
      } catch (OurException ae) {
         System.out.println("Caught it: " + ae);
      }
      System.out.println("Procedure first done.");
   }
   public static void second() throws OurException {
      third();
   }
   public static void third() throws OurException {
      throw new OurException("an error");
   }
}
```

```
public class Ex {
   public static void first() {
      try {
         second();
      } catch (RuntimeException ae) {
         System.out.println("Caught it: " + ae);
      }
      System.out.println("Procedure first done.");
   }
   public static void second() {
      third();
   }
   public static void third() {
      throw new Runtime("an error");
   }
}
```

Unchecked:
no throws needed

throws and inheritance

- Changes signature of function if checked exception uncatched
- If child class is overriding a parent class function with a signature that throws an exception, child class function needs to throw exception as well
- Summing up:
  - Case 1: Overriding method throws runtime exception (unchecked exception)- Allowed
  - Case 2:Overriding method throws subclass of the exception type declared by the parent method - Allowed
  - Case 3: Overriding method throws base class of exception type declared by the parent method - NOT Allowed
  - Case 4: Overriding method throws totally unrelated compile time exception replacing parent method throws clause - NOT Allowed

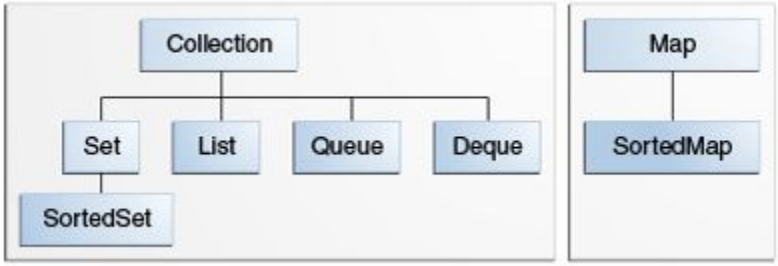
```
//What is wrong with this? And how would you fix it?
     class Example {
          public boolean writeToFile(String fileName, String data) {
               try
               {
                    File file = new File(fileName);
                    FileWriter fw = new FileWriter(file.getAbsoluteFile());
                    BufferedWriter bw = new BufferedWriter(fw);
                    bw.write(content);
                    bw.close();
                    return true;
               }
               catch (IOException e)
               {
                    return false;
               }
          }
     }
```

```
//TODO fix...
     class Example {
          public boolean writeToFile(String fileName, String data) {
               BufferedWriter bw = Null;
               try {
                    File file = new File(fileName);
                    //if file doesn’t exists. Then create it:
                    if (!file.exists())
                    {  file.createNewFile(); }
                    FileWriter fw = new FileWriter(file.getAbsoluteFile());
                    bw = new BufferedWriter(fw);
                    bw.write(content);
                    bw.close();
                    return true;
               }
               catch (IOException e) {
                    //bw.close(); check if this happens
                    return false;
               }
               finally
               {
               }
```

Collections and Generics

What are Java Collections?

- A collection is a data structure which contains and processes a set of data
- Data is encapsulated and access to data is only possible using predefined methods
  - Reduces programming effort
  - Increases speed
  - Reduces effort to learn new API
- Typical collections: stacks, queues, sets, lists and trees



Collections Example

```
import java.util.*;

public class ArrayToList
{
    public static void main(String[] args)
    {
        // This gives us nothing good
        System.out.println(args);

        // Convert args to a List of String
        List<String> argList = Arrays.asList(args);

        // Print them out
        System.out.println(argList);
    }
}
```

Generics and collections

```
Collection<String> c = ...
c.add(“Hello”)
c.add(“World”);
...
for (String s : c) {
  System.out.println(s.length + “ : “ + s.length());
```

A type like Collection<String> is called an instantiation of the parameterized type Collection.

Generics and collections

The compiler automatically detects uses of collections with incorrect types

```
Collection<String> c = ...
c.add(“Hello”) /* Okay */
c.add(1979);   /* Illegal: static error! */
```

An instantiation like Collection<String> behaves like the parameterized type Collection<T> where all the occurrences of T have been substituted with String

Subtyping and generic types

```
interface Collection<T> { ... }
interface List<T> extends Collection<T> { ... }
class LinkedList<T> implements List<T> { ... }
class ArrayList<T> implements List<T> { ... }

/* The following statements are all legal. */

List<String> l = new LinkedList<String>();
ArrayList<String> a = new ArrayList<String>();
Collection<String> c = a;
l=a;
c = l;
```

Subtyping extends to generic types

Generic Types

- Generics in Java provide a way to communicate T, the type of elements in a collection, to the compiler
- Compiler can check that you have used the collection consistently
- Result: safer and more-efficient code

Using Generic Types

- <T> is read, “of T”
  - For example, Stack<Integer> is read “Stack of integer”. Here the “T” is “Integer”
- The type annotation <T> informs the compiler that all extractions from this collection should be automatically cast to T
- Specify type declaration, can be checked at compile time
  - Can eliminate explicit casts

Example: Using Generics and Collections

```
class MyCollections
{
    public static <T> List<T> reverse(List<T> src)
    {
        List<T> results = new ArrayList<T>(src);
        Collections.reverse(results);
        return results;
    }
}
```

Advantage of Generics

- Declaring Collection<String> c tells us something about variable c (i.e. c holds only Strings)
  - Is true wherever c is used
  - The compiler checks this and won't compile code that violates this

- Without use of generic types, explicit casting must be used
  - A cast tells us something the programmer thinks is true at a single point in the code
  - The java virtual machine checks whether the programmer is right only at runtime

OCaml Program

```
let s x = x + 1

let rec i min max =
  if max < min then [] else min :: i (s min) max

let rec f p = function
 | []  -> []
 | a::r ->
   let fpr = f p r in if p a then a::fpr else fpr

let x n =
  f (fun m -> not (m mod n = 0))
```

- What are the types of the top-level OCaml identifiers s, f, x ?
- Evaluate x 10 [10; 31];;