Handling Errors

What should we do when something goes wrong?

Handling Errors

What should we do when something goes wrong?

- Bugs…
- User error…
- Bad input…
- Corrupted files…
- Unexpected results…

Handling Errors

What should we do when something goes wrong?

Option 1: Terminate

```
// Require: n >= 0
public static int factorial(int n) {
   if (n < 0) System.exit(0);
   assert(n >= 0);
   int ans = 1;
   for (int i=2; i<= n; i++) ans*=i;
   return ans;
}
```

Handling Errors

What should we do when something goes wrong?

Option 1: Terminate.

- Pros:
  - Halts immediately.
  - Clearly indicates a problem.
- Cons:
  - No attempt at recovery.
  - All cases treated identically.
  - Little debugging information provided.

Handling Errors

What should we do when something goes wrong?

Option 2: Print out an error.

```
// Require: n >= 0
public static int factorial(int n) {
   if (n < 0) {
         System.out.println("Error! Bad input.");
         return -1;
   }
   assert(n >= 0);
   int ans = 1;
   for (int i=2; i<= n; i++) ans*=i;
   return ans;
}
```

Handling Errors

What should we do when something goes wrong?

Option 2: Print out an error.

- Pros:
  - Provides some debugging information.

- Cons:
  - Program keeps running.
  - What value should be returned?
  - No indication of error to the program.
  - No mechanism for recovery.
  - What if your "user" is another program?

Handling Errors

What should we do when something goes wrong?

Option 3: Integrate into control flow.

```
// Require: k != null
public int insert(key k) {
   boolean success = true;

   ...

   if (success) return 1;
   else return -1;
}
```

Handling Errors

What should we do when something goes wrong?

Option 3: Integrate into control flow.

- Pros:
  - Returns information on errors.
  - Can provide specific error information.
  - Enables recovery.
- Cons:
  - Complicates main program. For example, does every method have to return an error status??

Exceptions

Goals:

- Indicates when an error has occurred.
- Stops execution on error.
- Simplifies recovery from errors by providing information that the calling **program** can use to recover.
- Minimal overhead when there are no errors.
- Simplifies debugging of errors.

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.

Exceptions

Construct an exception object:

- Exceptions are just objects:
  - Exception (base class for all exceptions)
  - ArithmeticException
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException
  - UnsupportedOperationException
  - FileNotFoundException

Exceptions

Two types of exceptions:

- Checked exceptions:
  - IOException
  - MySpecialException
- Runtime exceptions
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException

Exceptions

Construct an exception object:

- Exceptions are just objects
- Build a new exception object:

```
Exception e = new IllegalArgumentException(“Bad
   input: key should not be null.”);
```

Exceptions

Construct an exception object:

- Exceptions are just objects
- Build a new exception object.
- All exceptions extend class **Exception**.
- You can/should create your own exception types:

```
class LinkedListException extends Exception {
}
```

Exceptions

All exceptions support:

public class	**ExceptionClass**	
	ExceptionClass(String msg)	*Constructor*
String	getMessage()	*Returns message*
void	printStackTrace()	*Prints the call stack*

Exceptions

Construct an exception object:

- Exceptions are just objects
- Build a new exception object.
- All exceptions extend **Exception**.
- You can/should create your own exception types.

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.

Handling Errors

```
// Require: n >= 0
public int fact(int n) throws FactorialException
{
   if (n < 0) {
         throw new FactorialException("n < 0");
   }
   assert(n >= 0);
   int ans = 1;
   for (int i=2; i<= n; i++) ans*=i;
   return ans;
}
```

Exceptions

Throwing exceptions:

- On error, throw the exception!
- Method signature must indicate which (checked) exceptions it may throw.

Handling Errors

```
// Require: n >= 0
public static int factorial(int n)
   throws IllegalArgumentException,
          ArithmeticException,
          FactorialException
{
...
}
```

Exceptions

Throwing exceptions:

- On error, throw the exception!
- Method signature must indicate which (checked) exceptions it may throw.
- May throw many types of exceptions.

Handling Errors

```
interface MyExcellentInterface {

   int factorial(int n) throws FactorialException;

   void doWork(Foo f) throws IOException;

}
```

Exceptions

Throwing exceptions:

- On error, throw the exception!
- Method signature must indicate which (checked) exceptions it may throw.
- May throw many types of exceptions.
- Exceptions must be declared in the interface.

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
```

Exceptions

Handling exceptions:

- Wrap risky code in a **try block**.

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
catch (FactorialException e){
   // Oops, there was a problem.
   // Do something!
}
```

Exceptions

Handling exceptions:

- Wrap risky code in a **try block**.
- **Catch** your (checked) exceptions.

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
catch (FactorialException e){
   // Oops, there was a problem.
   // Do something!
}
finally {
   // Cleanup code
   // This code is executed in all cases.
}
```

Exceptions

Handling exceptions:

- Wrap risky code in a **try block**.
- **Catch** your (checked) exceptions.
- Put any clean-up code in a **finally** block.
  - Example: closing files
  - Example: completing initialization
  - Example: removing inconsistent states

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
catch (FactorialException e){
   // Oops, there was a problem.
   // Do something!
   // Terminate?
   // Print out an error?
   // Return an indicator?
   // Recover and continue?
}
```

Handling Errors

```
// Uh-oh, factorial function may throw an exception!
// We're doing something risky:
try {
   int j = factorial(n);
   System.out.println("Factorial was a success");
}
catch (FactorialException e){
 throw new Exception("Problem with factorial:" + e);
}
```

Handling Errors

```
void doWork(){   // Does not throw any exceptions

   // We're doing something risky:
   try {
         int j = factorial(n);
         System.out.println("Factorial was a success");
   }
   catch (FactorialException e){
         // Handle the exception here.
         // Do not pass it on!
   }

}
```

Exceptions

Indicating an error:

1. Construct an exception object.
2. Throw the exception.

Handling an error:

1. Catch the exception.
2. Recover.