Exception Handling and Functions as Data

Corky Cartwright

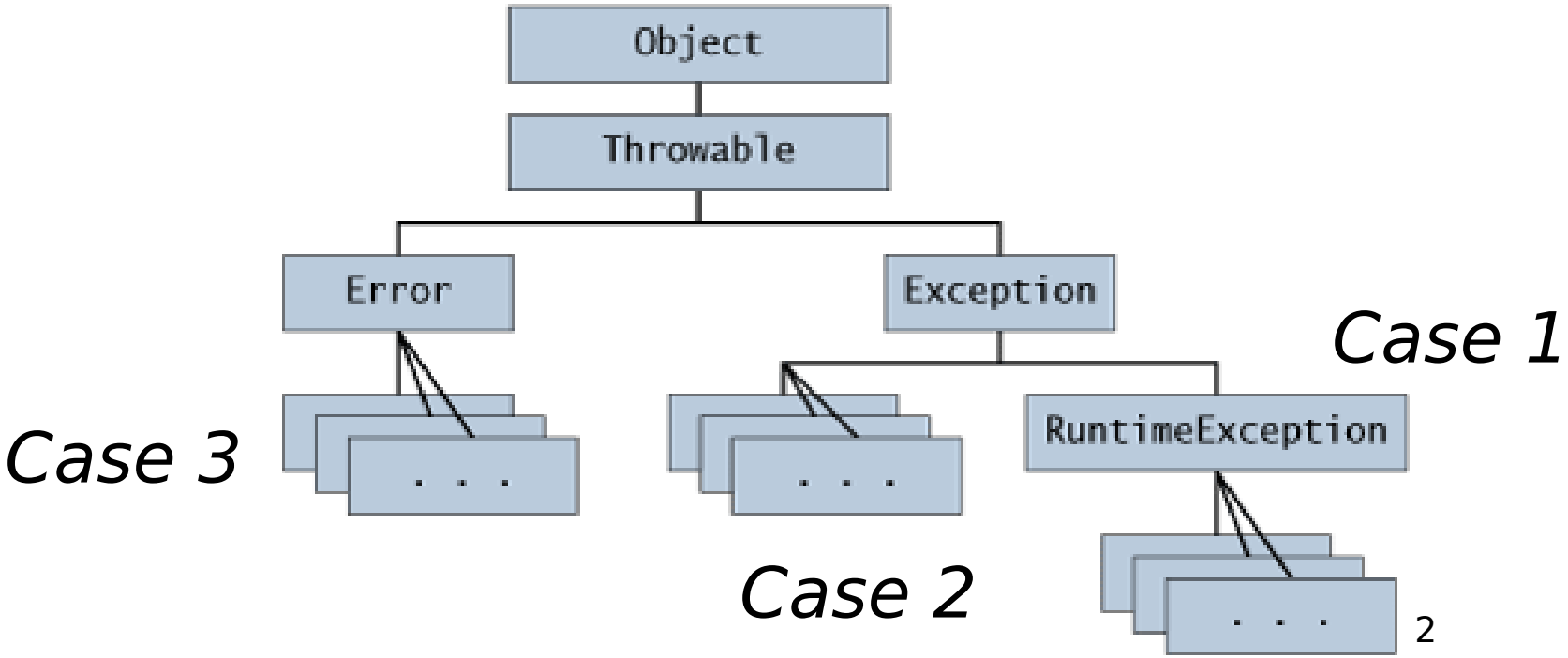
Stephen Wong

Department of Computer Science

Rice University

Errors and Exceptions in Java

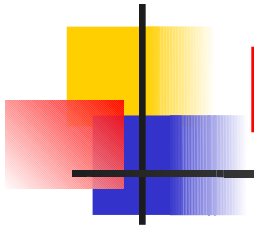
- In Java, the common supertype **`Throwable`** includes all error values and exception values.



Case 1: Subclass `RuntimeException`

- Used for error conditions that a program may want to handle, but are not part of a method's contract e.g.,
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `ArithmeticException` (*e.g.*, divide by zero)
  - `NegativeArraySizeException`
  - `ArrayStoreException`
  - `ClassCastException`
  - `IllegalArgumentException`

We will primarily use `RuntimeException` (Case 1) in this course except when the use of a library dictates the use of Case 2 or Case 3. In practice, a checked exception (Case 2) is a bad idea.



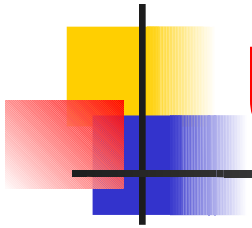
Example

Assume that we extend our `IntList` class hierarchy to include the method `rest()` in class `IntList` as follows:

```
/** IntList ::= EmptyIntList | ConsIntList(int, IntList). */
abstract class IntList {
  /** @return rest of this assuming this is non-empty. */
  IntList rest() { return ((ConsIntList) this).rest();
  /** Sorts this IntList into ascending (non-descending) order. */
  abstract IntList sort();
  /** Adds the int n to the front of this IntList. */
  IntList add(int n) { return new ConsIntList(n, this); }
  /** Inserts n in order, given this is sorted in ascending order. */
  abstract IntList insert(int n);
}
```

What does `EmptyIntList.ONLY.rest()` return?

A `ClassCastException`



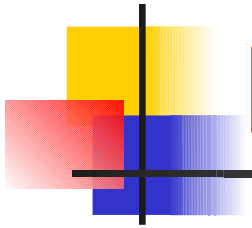
Unhandled Exceptions

- An Unhandled Exception results in program exit with a stack trace e.g.,

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at T1.foo(T1.java:50)

. . .
```

- The line numbers in the stack trace can help you locate the source of the error

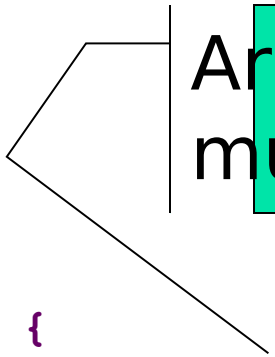


Handled Exceptions

- The programmer has the option of handling exceptions in Java with a try-catch statement. In most cases, unchecked exceptions correspond to coding errors. In large systems (like DrJava), it is common to have a top-level exception handler that logs the exception, perhaps updates the GUI to indicate that an error has happens, and recovers to the last valid program state.
- In some cases, the program may catch the exception near its source and return a value indicating failure or perform a failure action.

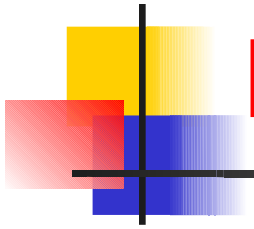
Throwing Exceptions

- The programmer also has the option of throwing instances of RuntimeException for user-defined errors e.g.,



Argument of throw statement must be of type Throwable

```
class T2 {
  int x;
  . . .
  float bar(int y) {
    if (y < 0) throw new ArithmeticException("Negative arg");
    n = y/x; // throws ArithmeticException if x = 0
    return n;
  }
}
```



Exception Objects

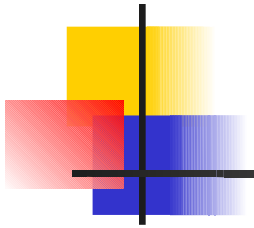
- In Java, exceptions are conventional objects, and can be created by expressions of the form

  new <exception-class>(<arg$_1$>, ..., <arg$_n$>)

- Examples

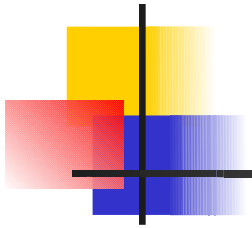
```
throw new IllegalArgumentException
("max applied to an empty list")

throw new java.util.NoSuchElementException
(”no more elements")
```



Type Casts and ClassCastException

- Java supports type casts (checks) for cases when the declared or inferred type of an expression is weaker than what is required for a particular computation.
- **`(<type>) <expr>`** simply converts the type of **`<expr>`** to **`<type>`** for type-checking purposes. If the value of **`<expr>`** does not have type **`<type>`**, the computation throws a **`ClassCastException`**.
- If the cast needs to be performed repeatedly, it is also possible to assign **`<expr>`** to a new variable declared to be of **`<type>`**.
- **Example**: consider the **`merge`** method on **`IntList`** for the current homework (HW7) written using the conventional Scheme solution. (This code is not a valid solution to the homework problem! In the homework, you must use dynamic dispatch instead of **`if`**.)

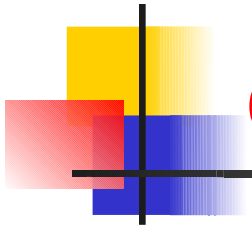


`merge` Example

```
abstract class IntList {
  IntList cons(int n) { return new ConsIntList(n, this); }
  abstract IntList merge(IntList other);
}

class EmptyIntList extends IntList {
  static EmptyIntList ONLY = new EmptyIntList();
  private EmptyIntList() { }
  IntList merge(IntList other) { return other;}
}

class ConsIntList extends IntList {
  int first;
  IntList rest;
  IntList merge(IntList other) {
    if (other == EmptyIntList.ONLY) return this;
    ConsIntList o = (ConsIntList) other; // cast operation
    if (first <= o.first()) return rest.merge(o).cons(first);
    else return merge(o.rest()).cons(o.first());
  }
}
```



Casting vs. Compiler Type-Checking

The type-checking in the Java compiler disallows casts

```
(<type>) <expr>
```

where **<type>** is an object type and the static type of **<expr>** and **<type>** do not overlap (other than **null**)

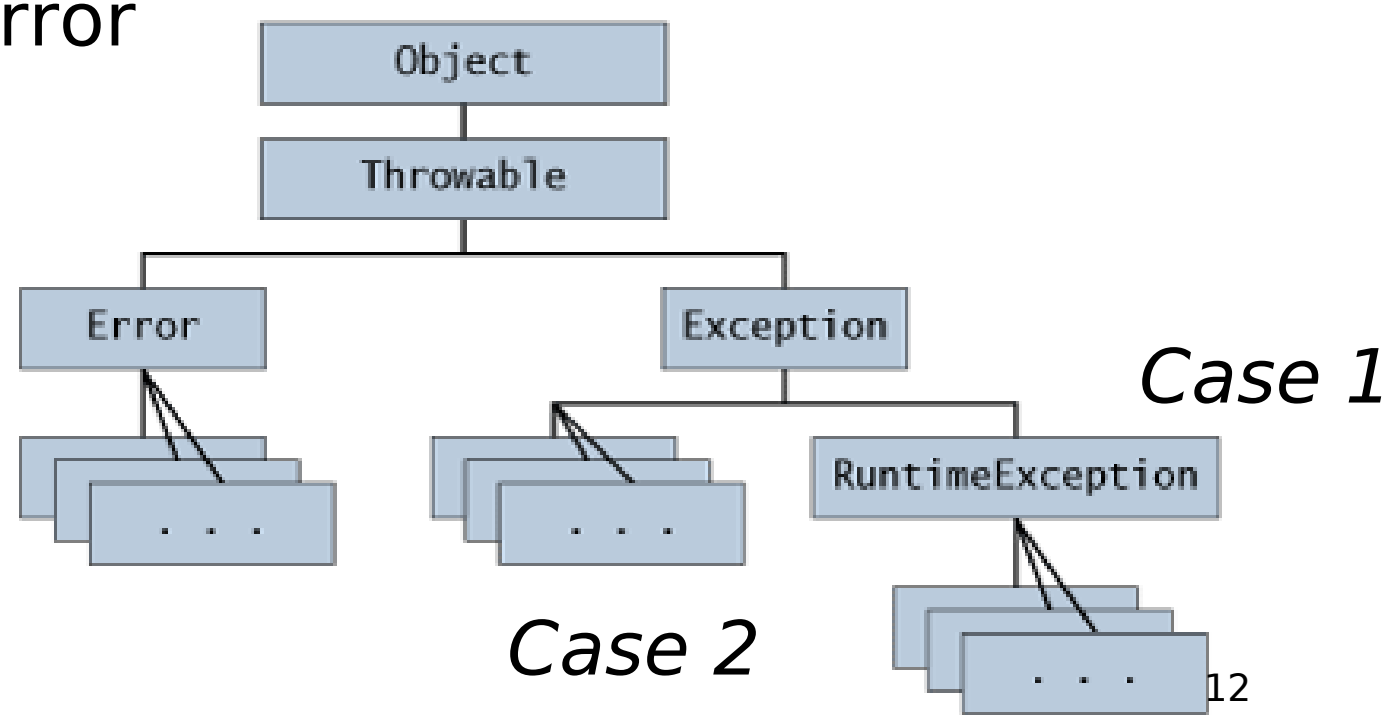
For example

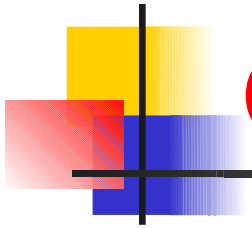
```
ConsIntList o = (ConsIntList) new EmptyIntList();
```

will result in a compile-time error

Cases 2 and 3

- Case 2: subtype of **`Exception`**, but not a subtype of **`RuntimeException`** (also called "checked exceptions")
- Case 3: Error



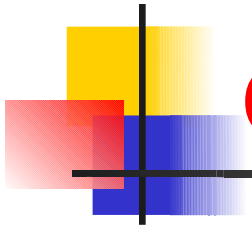


Case 2: Checked Exceptions

- Used for error conditions that a program may want to handle, and that are also explicitly part of a method's contract e.g.,

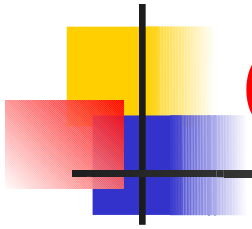
```
void foo() throws MyException { . . . }
```

- The Java compiler enforces the following rules on checked exceptions
  - Every method that throws a checked exception must advertise it in the throws clause in its method definition
  - Every method that calls a method that advertises a checked exception must either handle that exception (with try and catch) or must in turn advertise that exception in its own throws clause.



Checked Exceptions: a Bad Idea

- ML, a statically typed alternative to Scheme, was designed long before Java and includes an extensive exception facility. In ML ***all*** exceptions are unchecked. Why?
- If you include exceptions in the type system, program typing becomes very brittle. A trivial refactoring transformation or an insertion of simple debugging code (*e.g.*, to print a message to a file) can break type correctness. *This problem continually arises in developing Java programs.* When I defend Java as a good language for real world software development, my research colleagues (who only program in ML) jump on this issue. In these discussions, I concede that the designers of Java may have been stupid in some respects but still produced a decent language.



Case 3: Errors

- Subtypes of **`Error`** are used to identify error conditions that normal programs (including all your programs!) are not expected to handle.
- One direct subtype of **`Error`** is **`VirtualMachineError`**, which in turn includes the following direct subtypes
  - **`InternalError`**
  - **`OutOfMemoryError`**
  - **`StackOverflowError`**
  - **`UnknownError`**
- A **`VirtualMachineError`** is “thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating”

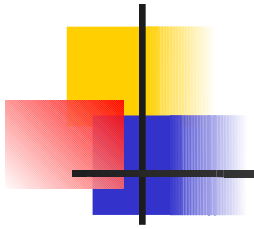
Encoding First-class Functions in Java

- Java methods are *not* data values; *they cannot be used as values.*
- But java classes include methods so we can indirectly pass methods (functions) by passing an appropriate class implementing an interface type that is designed exclusively to represent Java functions.
- Example: Scheme **`map`**

Interfaces for Representing Functions

For accurate typing, we need different interfaces for different arities. With generics, we can define parameterized interfaces for each arity. For now, we will have to define a loosely typed interface for each arity. Here is the code for **map**:

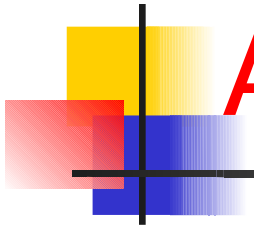
```
interface UnaryFun {
  Object apply(Object arg);  // Object -> Object
}
abstract class ObjectList {
  ObjectList cons(Object n) {
    return new ConsObjectList(n, this);
  }
  abstract ObjectList map(UnaryFun f);
}
  ...
```



Representing Specific Funcions

- For each function that we want to use a value, we must define a class, preferably a singleton. Since the class has no fields, all instances are effectively identical.
- Java provides a lightweight notation for singleton classes called *anonymous classes*. Moreover these classes can refer to fields and **`final`** method variables that are in scope.
- Anonymous class notation:

```
new <type>() {
   <member1>
   ...
   <membern>
}
```



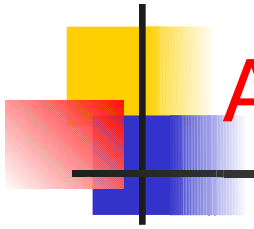
Anonymous Class Example

```
new UnaryFun() {
  Object apply(Object arg) {
      // Return a list containing arg
    return EmptyObjectList.ONLY.cons(arg);
  }
}
```

There are pending proposals to provide better notation for lambda abstractions.

Free Variables in Anonymous Classes

- What do free variables mean inside anonymous classes? What do they mean in λ-expressions?
- In Java, the free variables can be either:
  - fields, or
  - local (method) variables.
- Use them in doing the `filter` problem in HW8.



Another Anonymous Class Example

```
class FunUtils {
  UnaryFun compose(UnaryFun f, UnaryFun g) {
    return new UnaryFun() {
      Object apply(Object o) {
        return f.apply(g.apply(o));
  }
}
```