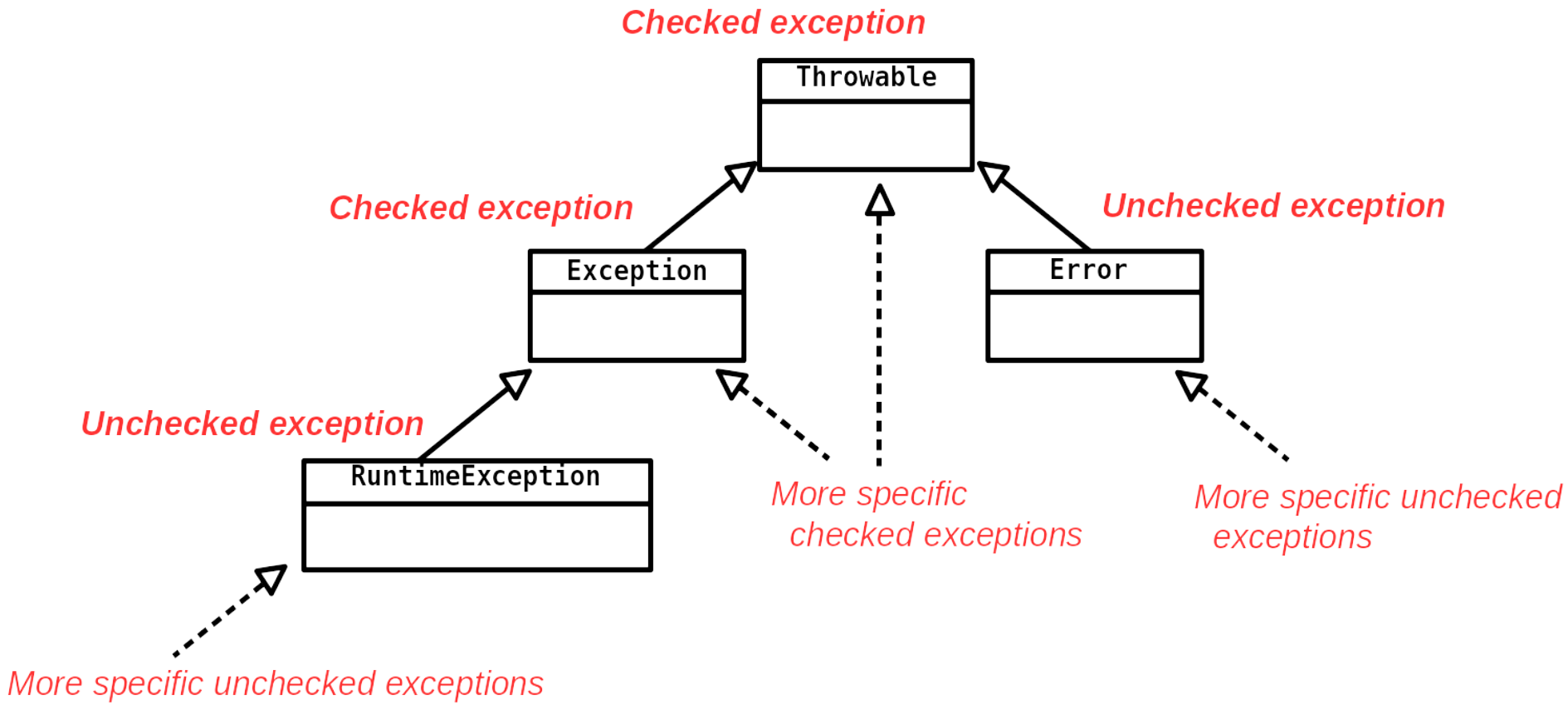
Unchecked and checked exceptions in Java

Exception classification

- Unchecked exceptions
  - errors: `Error` and their subclasses
    serious problems (e.g. `OutOfMemoryError`, `StackOverflowError`)
  - runtime exceptions: `RuntimeException` and their subclasses
    logic errors/precondition violations (e.g. `NullPointerException`, `IllegalArgumentException`)
- Checked exceptions: all other classes that are subclasses of `Exception` or `Throwable`
  Example: `java.io.IOException`
  In this case the user is forced to manage the exception in two ways:
  - either by handling the exception with **`try-catch`**
  - or by declaring that the constructor or method may throw the exception

Unchecked and checked exceptions in Java

Exception hierarchy



`throws` clauses

Exceptions can be declared in the headers of constructors and methods

Rules for checked exceptions

- Exception handling is enforced by the compiler for checked exceptions
- If the invocation of a constructor or method may throw a checked exception *E*, then
  - *E* is handled in the body with a **`try-catch`** (see `read1`)
  - or *E* is declared in the header (see `read2`)
- The static semantics forbids to catch a checked exception that can never be thrown

Example

```
static void read1(BufferedReader br) {
// does not throw or propagate checked exceptions
...
}
static void read2(BufferedReader br) throws IOException {
// could throw or propagate exceptions of type IOException
...
}
```

Error handling

The place where a failure occurs is often not the right point to handle it

Example 1: error handled as soon as possible

```
static void read1(BufferedReader br) {
    String line;
    do {
        try {
            line = br.readLine(); // may throw IOException
        } catch (IOException e) {
            System.err.println(e.getMessage());
            return;
        }
        if (line != null)
            System.out.println(line);
    } while (line != null); // if line == null then EOF has been reached
}
public void caller(BufferedReader br) {
    read1(br); // catching IOException here is a static error!
    ...
}
```

Error handling

The place where a failure occurs is often not the right point to handle it

Example 2: error better handled at an higher level

```
static void read2(BufferedReader br) throws IOException {
    String line;
    do {
        line = br.readLine(); // may throw IOException, 'throws' clause needed
        if (line != null)
            System.out.println(line);
    } while (line != null); // if line == null then EOF has been reached
}
public void caller(BufferedReader br) {
    try { // the caller has more control on method 'read'
        read2(br);
    } catch (IOException e) {
        System.err.println(e.getMessage());
        ... // asks the user another file to read
    }
    ...
}
```

try-catch-finally versus try-with-resources

try-catch-finally

a **`finally`** block is always executed at the end

Solution 1 with try-catch-finally

```
static void tryClose(Closeable c) {
   try {
      if (c != null) c.close(); // may throw IOException
   } catch (IOException e) { System.err.println(e.getMessage()); }
}
public static void main(String[] args) {
   BufferedReader br = null;
   try {
      br = tryOpen(args[0]);     // may throw FileNotFoundException
      System.out.println(br.readLine());    // may throw IOException
   } catch (IOException e) {     // FileNotFoundException ≤ IOException
      System.err.println(e.getMessage());
   } finally {                   // always executed, even in case of exception
      tryClose(br);              // 'br' must be declared before try-catch-finally
   }
}
```

try-catch-finally versus try-with-resources

try-catch-finally

a **`finally`** block is always executed at the end

Solution 2 with try-catch-finally

```
static void tryClose(Closeable c) throws IOException {
   if (c != null) c.close();
}

public static void main(String[] args) {
   BufferedReader br = null;
   try {
      try {
         br = tryOpen(args[0]); // may throw FileNotFoundException
         System.out.println(br.readLine()); // may throw IOException
      } finally { // always executed
         System.out.println("trying closing");
         tryClose(br); // may throw IOException
      }
   } catch (IOException e) {
      System.err.println(e.getMessage());
   }
}
```

try-catch-finally versus try-with-resources

try-with-resources (since Java 8)

automatically closes “resources” and handles all possible exceptions

Example with try-with-resources

```
public static void main(String[] args) {
   try (var br = tryOpen(args[0])) { // BufferedReader ≤ AutoCloseable
      System.out.println(br.readLine());      // may throw IOException
   } catch (IOException e) {
      System.err.println(e.getMessage());
   }
}
```

Remarks

try-with-resources: simpler code, method `tryClose` not needed!

try-catch-finally versus try-with-resources

declaration of resources used in the **try** block
- must be initialized, hence **var** can be used
- must be of type `AutoCloseable`

Example with try-with-resources

```
public static void main(String[] args) {
   try (var br = tryOpen(args[0])) { // BufferedReader ≤ AutoCloseable
      System.out.println(br.readLine());      // may throw IOException
   } catch (IOException e) {
      System.err.println(e.getMessage());
   }
}
```

catches `IOException` thrown by
- the initialization of resources
- the **try** block
- method `close()` automatically called on the declared resources

try-with-resources

Rules

- **`try`**`(...)` contains declarations of resources: local variables (as `bf`) declared and initialized, with scope extending as far as the try block
- the types of the resources must be subtypes of `AutoCloseable`
- resources are auto-closed (if non null) in the reverse order of initialization
- catch clauses manage also exceptions thrown during the initialization or automatic closing of resources

Decorator design pattern

In a nutshell

- a way to extend objects
- more flexible than inheritance: supports dynamic, multiple extensions of single objects
- a decorator wraps the object to be decorated, and delegates to it the execution of some methods

Examples

- `BufferedReader` : constructor `BufferedReader(Reader)` allows buffering of characters of readers for efficiency
- `PushbackReader`: constructor `PushbackReader(Reader)` allows characters read from readers to be pushed back
- `PrintWriter`: constructor `PrintWriter(Writer)` allows formatted printing for writers

Decorator design pattern

Example

```
public interface Shape {
   double perimeter();
   double area();
}
public abstract class ShapeDecorator implements Shape {
   private final Shape decorated;    // the object to be decorated

   protected ShapeDecorator(Shape decorated) {
     this.decorated = requireNonNull(decorated);
   }
   @Override
   public double perimeter() {
      return decorated.perimeter(); // delegation
   }
   @Override
   public double area() {            // delegation
      return decorated.area();
   }
   @Override
   public String toString() {        // delegation
      return decorated.toString();
   }
}
```

Decorator design pattern

Example

```
import java.awt.Color;
import static java.util.Objects.requireNonNull;

public class ColoredShape extends ShapeDecorator {

   private final Color color;

   public ColoredShape(Shape decorated, Color color) {
      super(decorated);
      this.color = requireNonNull(color);
   }
   @Override
   public String toString() {
      return super.toString() + " with color " + color;
   }
}
```

Decorator design pattern

Example

```
public class Circle implements Shape {
   private double radius;

   protected Circle(double radius) {
      if (radius <= 0)
         throw new IllegalArgumentException();
      this.radius = radius;
   }
   @Override
   public double perimeter() {
      return 2 * Math.PI * this.radius;
   }
   @Override
   public double area() {
      return Math.PI * this.radius * this.radius;
   }
   @Override
   public String toString() {
      return "a circle of radius " + radius;
   }
}
```

Decorator design pattern

Example

```
public class DecoratorTest {

   public static void main(String[] args) {
      Circle circle = new Circle(4);
      ColoredShape coloredCircle = new ColoredShape(circle, Color.blue);
      System.out.println(circle);
      // a circle of radius 4.0
      System.out.println(coloredCircle);
      // a circle of radius 4.0 with color java.awt.Color[r=0,g=0,b=255]
      assert circle.area() == coloredCircle.area();
      assert circle.perimeter() == coloredCircle.perimeter();
      circle = coloredCircle; // type error! ColoredShape not subtype of Circle
      Shape shape1 = circle; // ok
      Shape shape2 = coloredCircle; // ok
   }
}
```

Input/Output in Java

Main package `java.io`

- provides all basic features
- four parallel inheritance hierarchies:
  - input/output byte (binary) streams: `InputStream`, `OutputStream`
  - input/output char (text) streams: `java.lang.Readable` and `Reader`, `Writer`
- many classes implement the decorator design pattern to add extra features

More recent package `java.nio`

Other useful/advanced features

Convenient classes for input/output character streams

`java.io.BufferedReader`

- it is possible to read lines of characters with `readLine`
- it is only possible to decorate input character streams (type `Reader`)
- to decorate byte streams as `System.in`, decorator `InputStreamReader` must be created with constructor `InputStreamReader(InputStream in)`
  Example:

```
new BufferedReader(new InputStreamReader(System.in))
```

`java.io.PrintWriter`

- it is possible to print lines of characters with `println`
- many variants of available constructors
  - `PrintWriter(String fileName)` to open files directly from their file name
  - `PrintWriter(Writer out)` to decorate character streams
  - `PrintWriter(OutputStream out)` to decorate byte streams

Input character streams

Example

Utility methods for opening and reading text files

```
static BufferedReader tryOpen(String fileName) throws FileNotFoundException {
   if (fileName != null)
      return new BufferedReader(new FileReader(fileName));
   return new BufferedReader(new InputStreamReader(System.in));
}

static void read(BufferedReader br) throws IOException {
   String line;
   do {
      line = br.readLine();
      if (line != null) // null means EOF
         System.out.println(line);
   } while (line != null);
}
```

Method `equals`

Recap on `==` and `equals`

- predefined operator `==` on objects means reference equality
- method **`boolean`** `equals(Object)` defined in `Object`
  weaker notion of equality needed when objects may represent the same value even when have different references (=identities)

Definition of `equals` in `Object`

```
public class Object {
...
   public boolean equals(Object obj) {
      return (this == obj); // at this abstract level == and equals() coincide
   }
...
}
```

Method `equals`

Typical example: string objects

```
String s1 = "a string";
String s2 = s1;
String s3 = new String(s1); // copy constructor
StringBuilder sb = new StringBuilder(s1);
assert s1 == s2;
assert s1 != s3;
assert s1.equals(s2);
assert s1.equals(s3);
assert !s1.equals(sb); // a string is not a string builder
```

A correct redefinition of `equals` and `hashCode`

Example for shapes

```
import static java.util.Objects.hash; // computes efficient hash codes

public class Circle implements Shape {
   private double radius;

   // omitted code

   @Override // redefines 'equals()' of 'Object'
             // 'final' means cannot be redefined in subclasses
   public final boolean equals(Object obj) {
     if (this == obj)
         return true;
     if (obj instanceof Circle c)
         return radius == c.radius;
     return false;
   }
   @Override // redefines hashCode()' of 'Object'
   public final int hashCode() {
     return hash(radius);
   }
}
```

Problems with `equals` and `hashCode`

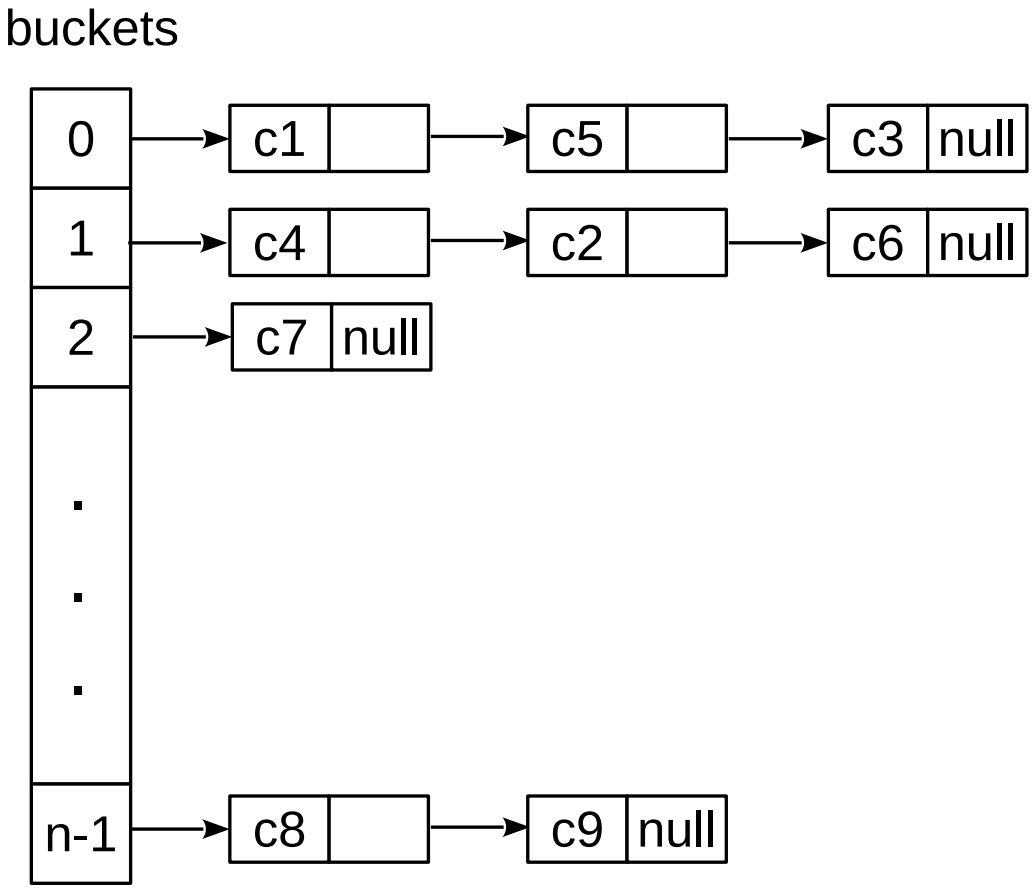
Remark

- **`boolean`** `equals(Object)` and **`int`** `hashCode()` defined in `Object`
- `equals` and `hashCode` are pervasively used to implement hash tables (example `HashSet<E>`)
- important: never override `equals` without overriding `hashCode`

Example of incorrect behavior when `hashCode` not redefined

```
import java.util.HashSet;
class HashSetTest {
  public static void main(String[] args) {
    var shapeSet = new HashSet<Shape>();
    var c1 = new Circle(2);
    var c2 = new Circle(2);
    shapeSet.add(c1);
    assert c1.equals(c2);
    // most likely fails if hashCode() is not redefined in  Circle
    assert shapeSet.contains(c2);
  }
}
```

A simple implementation of `HashSet` (1)




```
hash(c1)==hash(c5)==hash(c3)==0
hash(c4)==hash(c2)==hash(c6)==1
hash(c7)==2
```

A simple implementation of `HashSet` (2)

Example

```
public interface Set<E> {
   int size();
   boolean isEmpty();
   boolean contains(E element); // in 'java.util.Set' 'element' has type 'Object'
   boolean add(E element);      // returns 'true' iff the set has been changed
   boolean remove(E element);   // returns 'true' iff the set has been changed
}

import java.util.ArrayList;
import java.util.LinkedList;
public class HashSet<E> implements Set<E> {
    static final int DEFAULT_CAPACITY = 16;
    private int size;
    private final ArrayList<LinkedList<E>> buckets;
    private int capacity;
    public HashSet(int capacity) {
        if(capacity<0) throw new IllegalArgumentException();
        this.capacity = capacity;
        buckets = new ArrayList<>(capacity);
        for (int i = 0; i < capacity; i++) {
            buckets.add(new LinkedList<E>()); // appends a new empty bucket
        }
    }
    public HashSet() { this(DEFAULT_CAPACITY); }
```

A simple implementation of `HashSet` (3)

Example

```
    private int hash(E e) { // uses 'int hashCode()'
        return Math.abs(e.hashCode() % capacity); // '%' = reminder operator
    }
    public int size() { return size; }
    public boolean isEmpty() { return size == 0; }
    public boolean contains(E element) {
        return buckets.get(hash(element)).contains(element);
    }
    public boolean add(E element) {
        var b = buckets.get(hash(element));
        if (b.contains(element))
            return false;
        size++;
        return b.add(element); // appends the new element
    }
    public boolean remove(E element) {
        var removed = buckets.get(hash(element)).remove(element);
        if (removed)
            size--;
        return removed;
    }
}
```

HashMap: same problems as HashSet

Example of incorrect behavior when hashCode not redefined

```
import java.util.HashMap;
import java.awt.Color;

class HashMapTest {
    public static void main(String[] args) {
        var hm = new HashMap<Shape, Color>();
        hm.put(new Circle(2), Color.blue);
        // most likely throws NullPointerException
        // if 'hashCode()' is not redefined in Circle
        assert hm.get(new Circle(2)).equals(Color.blue);
    }
}
```

General rule to avoid these problems

If two objects o_1 and o_2 are equal according to `equals()`, then o_1`.hashCode()` must be equal to o_2`.hashCode()`

Problems with `equals`

Problems

- redefinition in subclasses may invalidate symmetry or transitivity
- redefinition may be incorrect if it depends from mutable fields

Safer solution, although drastic

- when redefined in non abstract classes, method `equals` (and `hashCode`) should be **`final`**
- if the behavior of `equals` must change, then the decorator pattern should be used

Final methods and classes

- **`final`** classes and interfaces cannot be extended
- **`final`** methods cannot be redefined in subclasses

`equals` and `hashCode` and the decorator pattern

Example (part 1)

```
public abstract class ShapeDecorator implements Shape {
   private final Shape decorated; // the object to be decorated
   ...
   @Override
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj instanceof ShapeDecorator sd)
         return decorated.equals(sd.decorated);
      return false;
   }
   @Override
   public int hashCode() { // delegation
      return decorated.hashCode();
   }
}
```

`equals` and `hashCode` and the decorator pattern

Example (part 2)

```
public class ColoredShape extends ShapeDecorator {
   private final Color color;
   ...
   @Override
   public final boolean equals(Object obj) {
      return super.equals(obj) && (obj instanceof ColoredShape cs) &&
        color.equals(cs.color);
   }

   @Override
   public final int hashCode() {
      return hash(super.hashCode(), color.hashCode());
   }
}
public class DecoratorTest {
   public static void main(String[] args) {
      var circle = new Circle(4);
      var coloredCircle = new ColoredShape(circle, Color.blue);
      assert !circle.equals(coloredCircle) && !coloredCircle.equals(circle);
      assert coloredCircle.equals(new ColoredShape(new Circle(4), Color.blue));
      assert !coloredCircle.equals(new ColoredShape(new Circle(5), Color.blue));
      assert !coloredCircle.equals(new ColoredShape(new Circle(4), Color.red));
   }
}
```