M1 (b) – Encapsulation

Jin L.C. Guo

Lab Test

- Send your email to Shi Yan Du (shi.du@mail.mcgill.ca)
- Title: **[COMP303]Lab Test SignUp**
- Include both Week and Time of the slot
- Check availability in advance

Recap of last class

- Information Hiding and Encapsulation
- Scope and Visibility
- Object Diagrams
- Escaping Reference
- Immutability

Recap of last class

- Activity
  - Add Color attribute to Card

```java
/**
* A card's suit.
*/
public enum Suit
{
    CLUBS, DIAMONDS, SPADES, HEARTS;

    public enum Color {BLACK, RED}

    public Color getColor()
    {
        switch(this)
        {
            case CLUBS:
                return Color.BLACK;
            case DIAMONDS:
                return Color.RED;
            case SPADES:
                return Color.BLACK;
            case HEARTS:
                return Color.RED;
            default:
                throw new AssertionError(this);
        }
    }
}
```

```java
/**
* A card's suit.
*/
public enum Suit
{
    CLUBS(Color.BLACK),
    DIAMONDS(Color.RED),
    SPADES(Color.BLACK),
    HEARTS(Color.RED);

    private Color aColor;

    public enum Color {BLACK, RED}

    Suit(Color pColor)
    {
        this.aColor = pColor;
    }

    public Color getColor()
    {
        return this.aColor;
    }
}
```

package-private/private access

Objectives of this class

- Design by Contract
- Assertion
- Documentation
- Code Style

Well Encapsulated Card Class

```
public class Card
{
    final private Rank aRank;
    final private Suit aSuit;

    public Card(Rank pRank, Suit pSuit)
    {
        aRank = pRank;
        aSuit = pSuit;
    }

    public Rank getRank()
    {
        return aRank;
    }

    ……
}
```

```
Card card1 = new Card(null, Suit.CLUBS);

System.out.println(card1.getRank().toString());
```

Things can still go wrong!

Contract (Human Interaction)

Supplier

Right
Responsibility

Client

Right
Responsibility

Design by Contract

- Documenting rights and responsibilities of software modules to ensure program correctness

Specify the interface

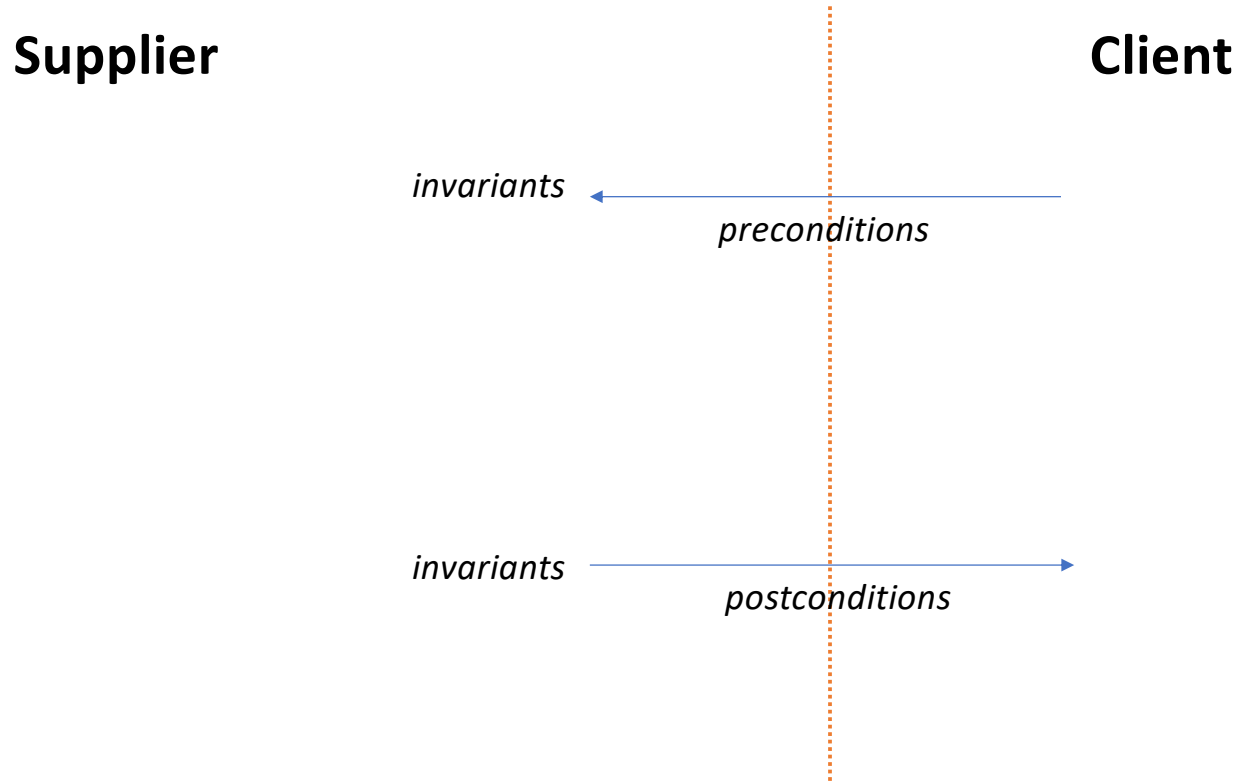
- Precondition – What must be true in order for the routine to be called

Client's responsibility

- Postcondition – What the routine is guaranteed to do; the state of the world when the routine is done.

Supplier's responsibility

- Class invariants – Conditions that's always true


Supplier
Client
invariants
preconditions
invariants
postconditions

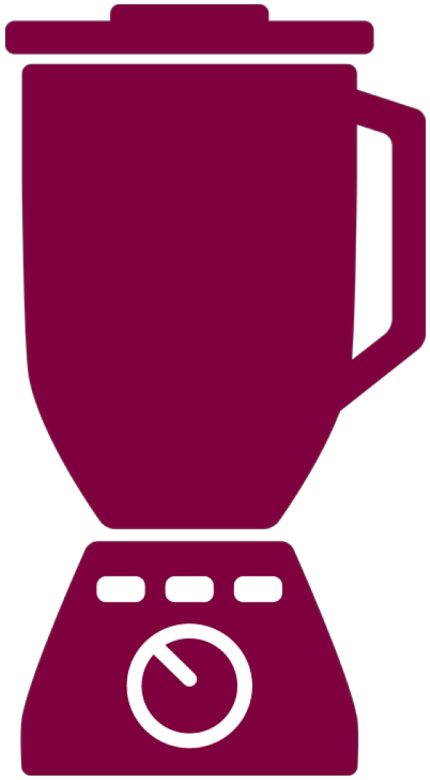
Specify Contract

```
/**
* @invariant getRank() != null && getSuit() !=null
*/



/**
* … …
* @pre pRank != null && pSuit != null
* @post getRank() == pRank && getSuit() == pSuit
*/
public Card(Rank pRank, Suit pSuit)
{
     aRank = pRank;
     aSuit = pSuit;
}
```

Activity 1

- Design an interface to a kitchen blender. It has ten speed settings (0-9, 0 means off). You can only operate when it's full. You can change the speed only one unit at a time (that is, from 0 to 1, and from 1 to 2, not from 0 to 2). Add appropriate pre- and postconditions and class invariant.

```
int getSpeed()
void setSpeed(int pSpeed)
boolean isFull()
void fill()
void empty()
```



```
/*
* @invariant if(getSpeed() >0) isFull()
* @invariant getSpeed()>=0 && getSpeed<10
*/


/*
* @pre Math.abs(getSeepd() - pSpeed) == 1
* @pre pSpeed>=0 && pSpeed<10
* @post getSpeed() == pSpeed
*/
void setSpeed(int pSpeed)


/*
* @pre !isFull()
* @post isFull()
*/
void fill()
```

similar with empty()

Verifying Contract

- No build-in support in Java
- Partially achieved by assertion

Java Assertions

assert *Expression1* ;

assert *Expression1* : *Expression2* ;

if *Expression1* is false throws an AssertionError

Safety-net, not enforcement!

Ensure things shouldn't happened won't happen

java –ea runs Java with **a**ssertions **e**nabled (disabled by default)

(Partially) Verifying Contract in Java

```
/**
* … …
* @pre pRank != null && pSuit != null
* @post getRank() == pRank && getSuit() == pSuit
*/
public Card(Rank pRank, Suit pSuit)
{
    assert pRank != null && pSuit != null;
    aRank = pRank;
    aSuit = pSuit;
    assert getRank() == pRank && getSuit() == pSuit;

}
```

(Partially) Verifying Contract in Java

- Evaluate the following contract for a stack class

```
/**
* … …
* @pre pCard != null
* @post pop() == pCard
*/
public void push(Card pCard)
{… …}
```

pop() -> peek()

Heisenbug

a software bug that seems to disappear or alter its behavior when one attempts to study it.

Heisenberg

Exceptions

- For exceptional conditions

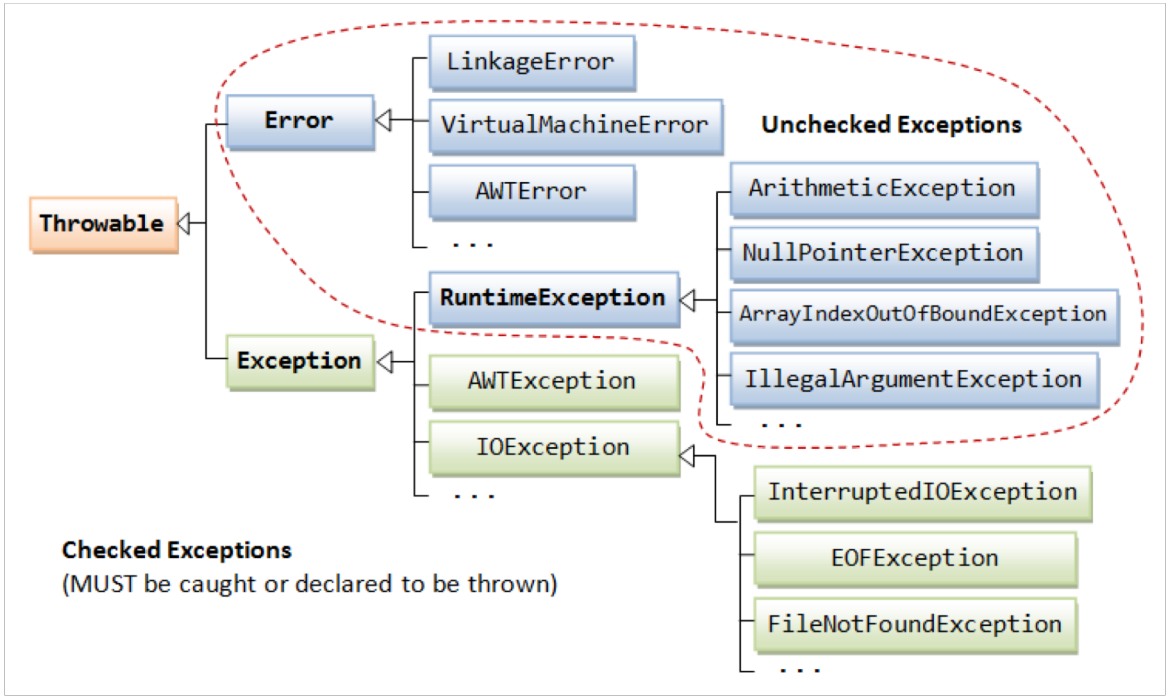
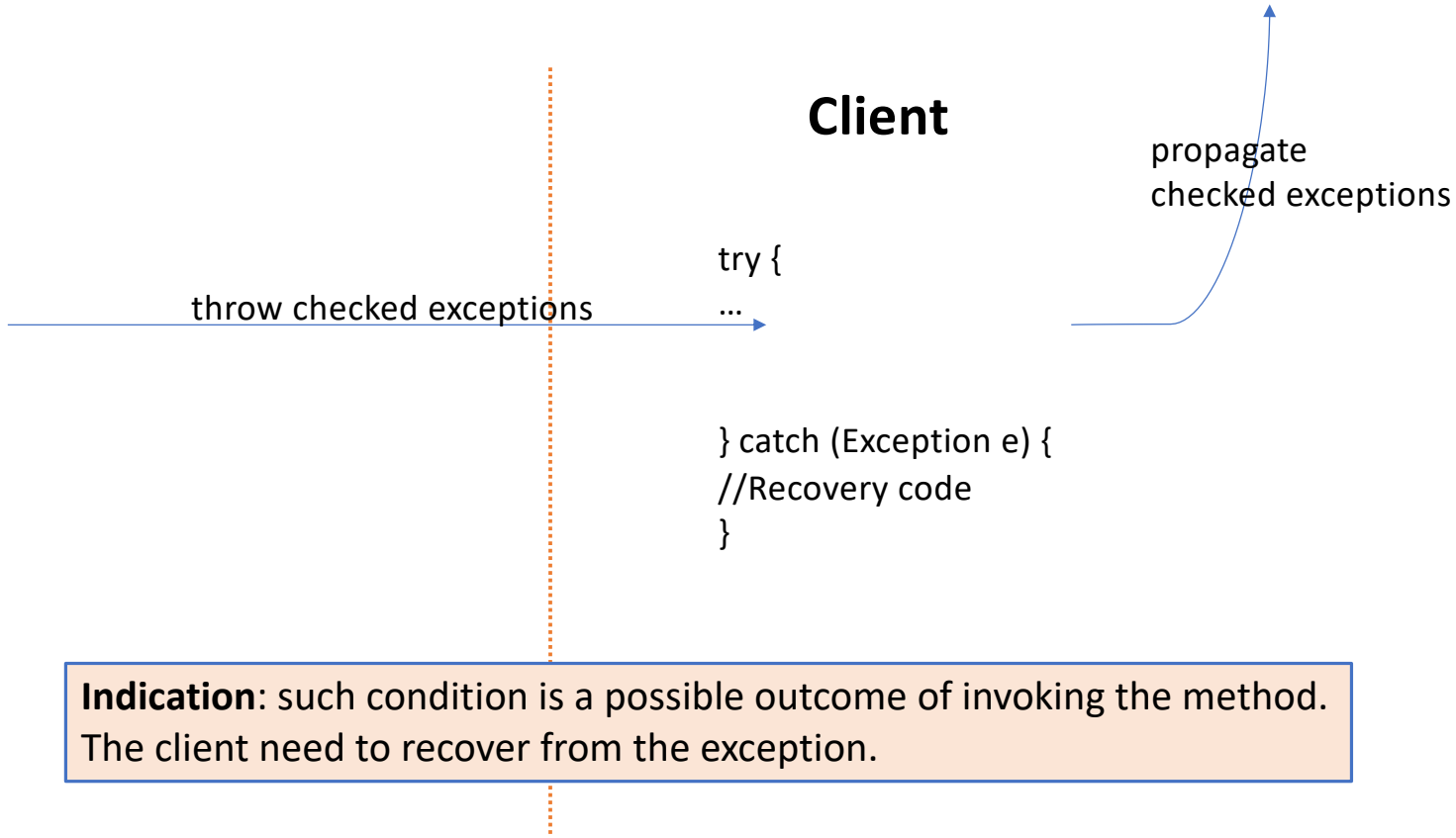
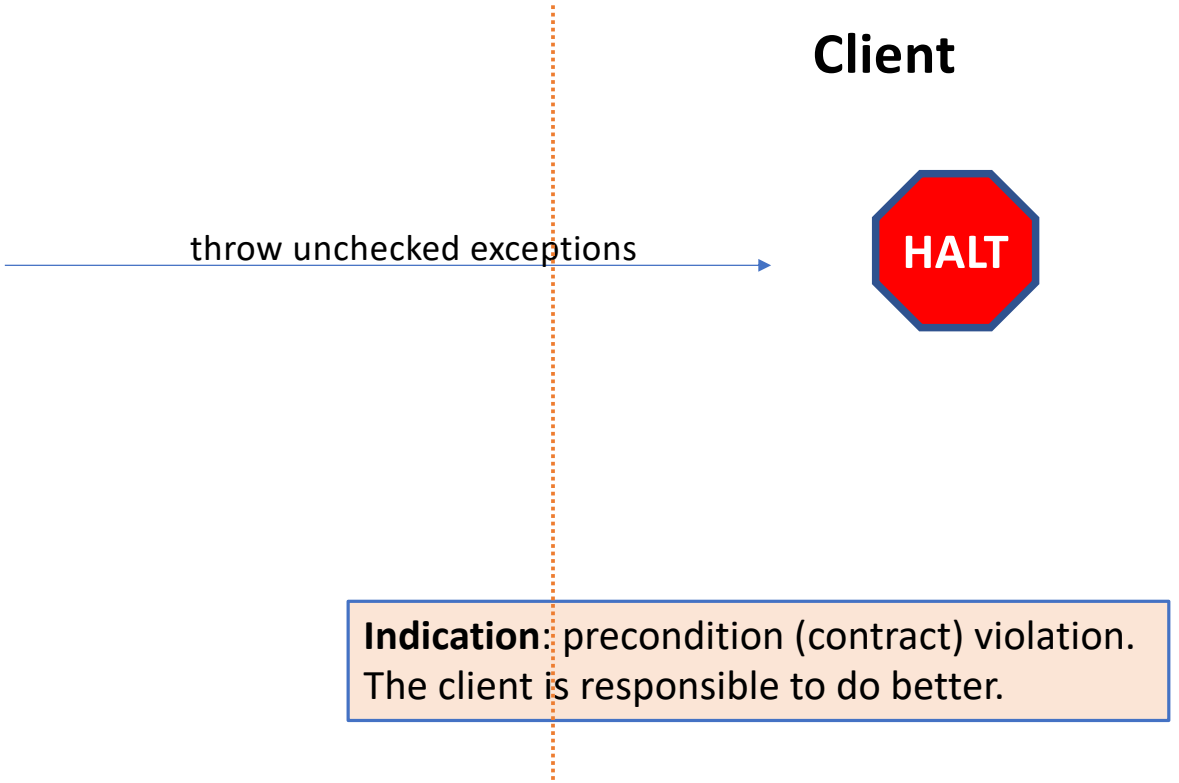


image source:http://www.ntu.edu.sg/home/ehchua/programming/java/images/Exception_Classes.png

Checked Exceptions



Runtime Exceptions



Exception for Contract Precondition

- For *public* methods, enforce precondition by explicit checks that throw particular, specified exceptions

```
/**
* @param pRank The index of the rank in RANKS
* @param pSuit The index of the suit in SUITS
* @throws IllegalArgumentException if pRank != null or pSuit != null
*/
public Card(Rank pRank, Suit pSuit)
{
     if( pRank == null || pSuit == null )
     {
          throw new IllegalArgumentException();
     }
     aRank = pRank;
     aSuit = pSuit;
}
```

Design by Contract

- Purpose: ensure program correctness
- Correct -> does no more and no less than it claims to do
- Being “lazy”: be strict in what you will accept before you begin, and promise as little as possible in return

- Benefit: forces the issue of requirements and guarantees at design time – what your code (doesn’t) promise to deliver
- Means: documenting and verifying

Documentation

- Interface
  - a comment block precedes the declaration of a class, data structure, or method.
- Data fields
  - a comment next to the declaration of a static or non-static variable.
- Implementation comments
  - a comment inside a method

Interface Documentation

- Define abstractions
- Information for using a class or method

Interface Documentation

- Define abstractions
- Information for using a class or method

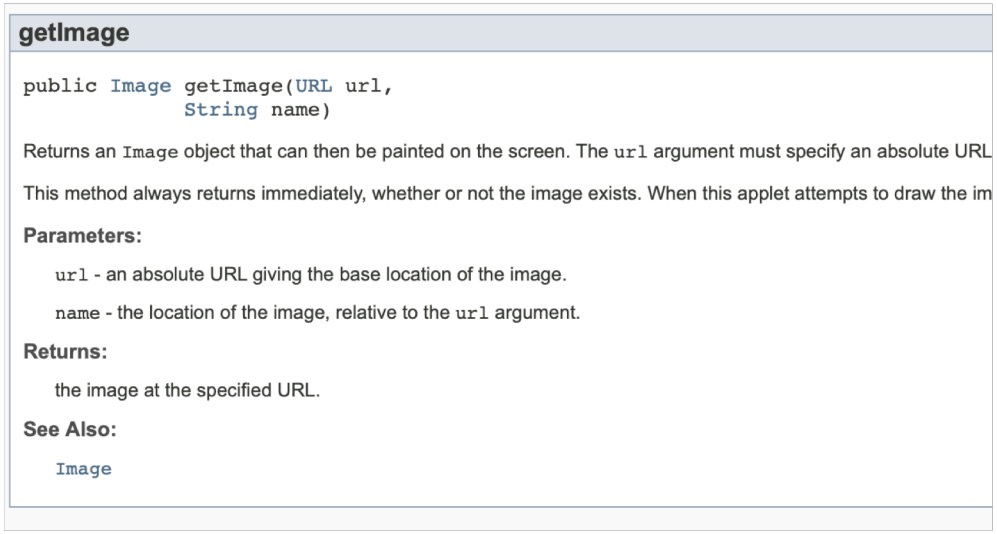
The comment doesn't do any of those!

```
/**
* Returns an Image object by their url
*
* @param url image url
* @param  name image name
* @return      image object
*/
public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
}
```

```java
/**
* Returns an Image object that can then be painted on the screen.
* The url argument must specify an absolute {@link URL}. The name
* argument is a specifier that is relative to the url argument.
* <p>
* This method always returns immediately, whether or not the
* image exists. When this applet attempts to draw the image on
* the screen, the data will be loaded. The graphics primitives
* that draw the image will incrementally paint on the screen.
*
* @param url   an absolute URL giving the base location of the image
* @param  name the location of the image, relative to the url argument
* @return      the image at the specified URL
* @see         Image
*/
public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
}
```

Use Javadoc for Public APIs

- Documentation -> HTML pages describing the classes, interfaces, constructors, methods, and fields.



getImage

```
public Image getImage(URL url,
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the im

Parameters:

`url` - an absolute URL giving the base location of the image.

`name` - the location of the image, relative to the `url` argument.

Returns:

the image at the specified URL.

See Also:

`Image`

Use Javadoc for Public APIs

- @param
- @return
- @throws
- @see
- @author
- {@code}

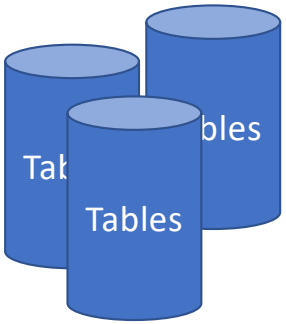
… …

@custom.mytag

javadoc -tag custom.mytag:a:"This is my Tag:”

Activity 2

- IndexLookup class for distributed storage system.



Object	Name	Age	…
A-1	John	20	…
A-2	Elizabeth	21	…
…	…	…	…

```
IndexLookup query = new IndexLookup(table, index, key1, key2);
Iterator iterator = query.iterator();
while(iterator.hasNext())
{
    object = iterator.next()
    … …
}
```

Activity 2

- Does the user needs to know the following:

  1. The format of message that **IndexLookup** class sends to the servers holding indexes and objects.

  2. The comparison function used to determine whether a particular objects falls in the designed range (comparison using integers, floating points, or strings)

  3. The data structure used to store indexes on servers

  4. Whether **IndexLookup** issues multiple requests to different servers concurrently

  5. The mechanisms for handling server crashes.

Data field

- Explain, not repeat

```
/**
* the horizontal padding of each line in the text
*/
private static final int textHorizontalPadding = 4;
```

```
/**
* The amount of blank space to leave on the left and
* right sides of each line of text, in pixels.
*/
private static final int textHorizontalPadding = 4;
```

Data field

- Fill in missing details (that you cannot get from name and type)

```
//Contains all term within the document and their number of
appearances
private TreeMap<String, Integer> termAppearances;
```

```
//Hold the statistics about the term appearances within a
//document in the form of <term, count> where the term is the
//word in its dictionary form, and the count is how many times
//it matches the tokens in the document after preprocessing.
//If a term doesn't match any token in the document, then
//there's no entry for that term.
private TreeMap<String, Integer> termAppearances;
```

Implementation comments

- For understand what the code is doing
  - Add a comment before each major block for abstract description

```
// Compute the standard deviation of list elements that are
// less than the cutoff value.
for (int i = 0; i < n; i++) {
      ...
 }
```

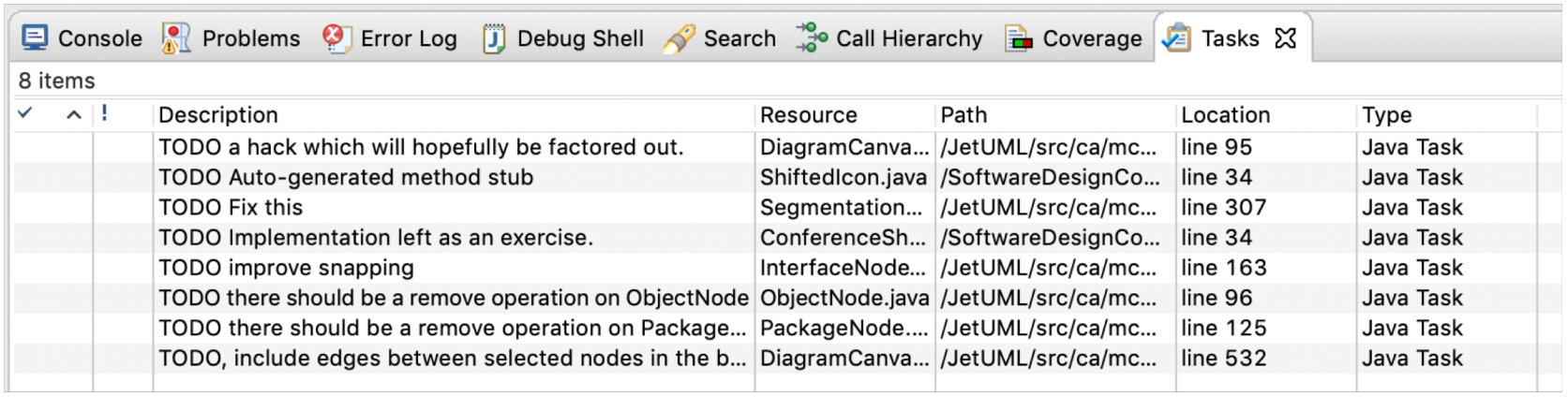
- For understand why the code is written this way.

```
// Arbitrary default value, used to simplify the testing code
private static final int DEFAULT_DIMENSION = 1000;
```

More Informative Comments

- *Record Assumptions*
- *Record Limitations*
- *TODO comments*

……



Smells in Comments

Repeat the code

About the implementation details

Journal comments

Misleading comments

Outdated comments

… …

Comments As a Design Tool

Write comments first:

- Capture the abstraction before implementation
- Reveal potential problem of design (complexity)
- Improve quality of documentation

Code Style

- Goal: reduce complexity
  - to understand the code
  - to make future changes

Naming Entities

- Packages
- Classes/Enums
- Interfaces/Annotations
- Members of Reference types
- Parameters
- Local variables

Naming Entities

- Principle
  - Be clear and descriptive
  - Reveal your intention
  - Follow conventions
    - Java Naming Conventions
    - EJ3: 68

```
int d; // elapsed time in days
```

```
int elapsedTimeInDays;
```

Formatting

- Braces
- Indentation
- Spacing

…

```
public class CommentWidget extends TextWidget
{
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";
  public CommentWidget(ParentWidget parent, String text){super(parent, text);}
  public String render() throws Exception {return “”; }
}
```

Not Easy to read…

Formatting

- Braces
- Indentation
- Spacing

…

Easy to read
Consistent

```
return new MyClass() {
  @Override public void method() {
    if (condition()) {
      try {
        something();
      } catch (ProblemException e) {
        recover();
      }
    } else if (otherCondition()) {
      somethingElse();
    } else {
      lastThing();
    }
  }
};
```

Style used in the exercises

- Access here.
- Braces
- pParam, and aParam is not about type enforcement, but a clear indication of the variable scope and semantic kind.
- Make principled changes as you need.
- Read more about style and cleanness in article *Making wrong code look wrong*.

Code Exploration

In JetUML

- Demension Class

- Rectangle Class

Acknowledgement

- Some examples are from the following resources:
  - *COMP 303 Lecture note* by Martin Robillard.
  - *The Pragmatic Programmer* by Andrew Hunt and David Thomas, 2000.
  - *Effective Java* by Joshua Bloch, 3rd ed., 2018.
  - *Clean Code* by Robert C. Martin, 2009
  - *A Philosophy of software design* by John Ousterhout, 2018