

# Algoritmer och Datastrukturer

## Föreläsning 8



Felhantering, säkerhet och  
best practices

# Dagens föreläsning

{

Vi ska:

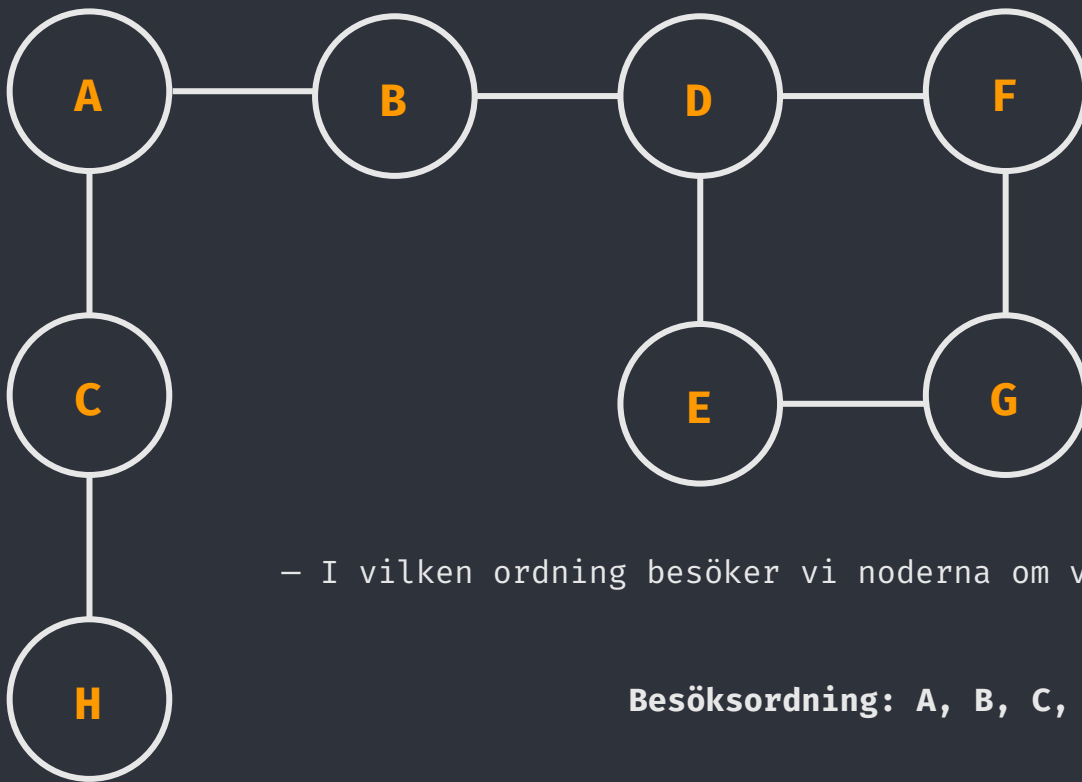
- Prata om best practices i Java som är bra att tänka på
- Kika på hur man skriver bättre och säkrare kod
- Diskutera undantagsfel och hur de bör fångas
- Kolla på vad som händer med stacken när undantag kastas, och hur vi bör kommunicera dem till andra utvecklare
- Lära oss hur man skriver sina egna exceptionklasser och varför det kan vara värdefullt

}

# Försök besvara dessa efter dagens pass

- 1) Vad är en rekursiv datastruktur för någonting?
- 2) Vilken tidskomplexitet har en sökning i ett rödsvart träd? Varför?
- 3) Vad är risken med obalanserade binära träd?
- 4) Ge exempel på några datastrukturer i Java som använder sig av balanserade träd.
- 5) Vad är en graf?
- 6) Ge några exempel på användningsområden för grafer.
- 7) Vad är garbage collection för någonting? Varför används det i Java?
- 8) Vad är några av fördelarna med en enhanced for loop?
- 9) Vad är några av nackdelarna?
- 10) Vilka är de tre vanliga sätten att traversera ett binärt sökträd? Vad är skillnaden mellan dem?
- 11) Vad är Dijkstras algoritm för någonting? Hur fungerar den?
- 12) Vad är undantag och hur kan man hantera dem?
- 13) Vad är stack unwinding (stackupplindning) för någonting?
- 14) Vad är skillnaden mellan checked och unchecked exceptions (märkta och omärkta undantag)?
- 15) Vad är tvåstegsinitialisering och varför ska man undvika det?
- 16) Vad är en girig algoritm? Hur fungerar den?

# Repetition: Bredden Först-sökning i graf

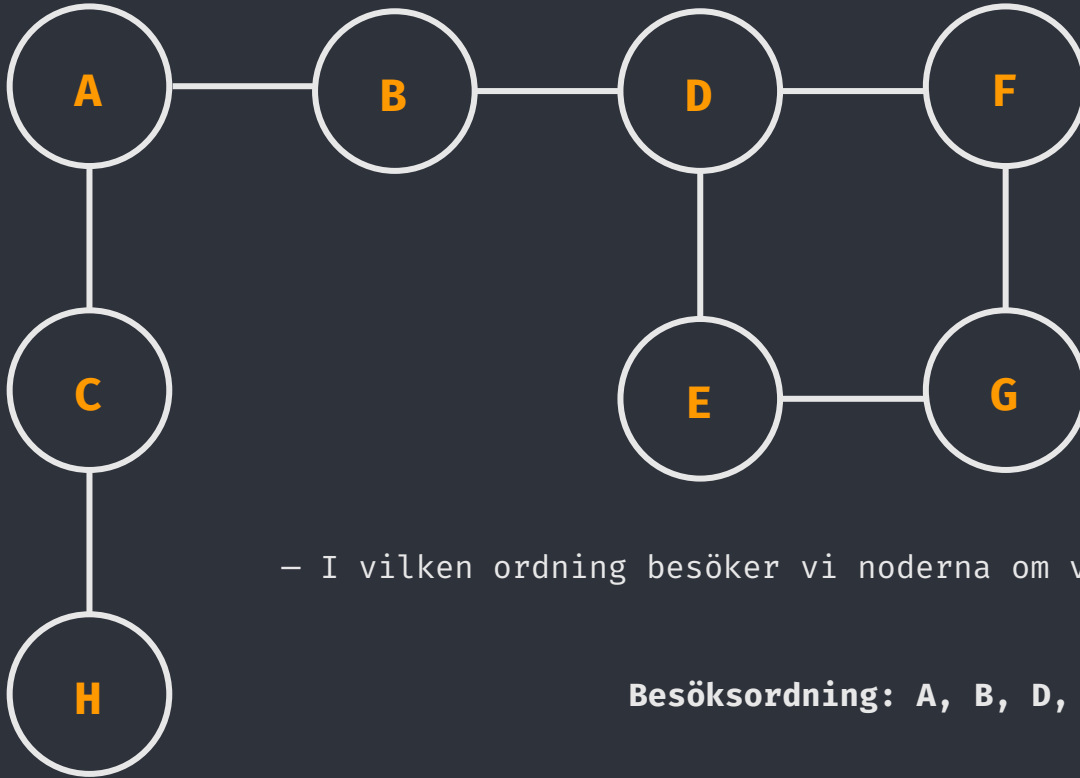


Använd en **kö**

– I vilken ordning besöker vi noderna om vi börjar från A?

**Besöksordning: A, B, C, D, H, E, F, G**

# Repetition: Djupet Först-sökning i graf



Använd en **stack**

– I vilken ordning besöker vi noderna om vi börjar från A?

**Besöksordning: A, B, D, E, G, F, C, H**

# Repetition: vad blir tidskomplexiteten?

- Vad blir tidskomplexiteten för själva for-loopen?

```
int[] keys = { 1, 2, 3, 4, 5 };  
String[] values = { "One", "Two", "Three", "Four", "Five" };  
  
TreeMap<Integer, String> map = new TreeMap<>();  
  
for (int n : keys)                                O(n) *  
{  
    map.put(keys[n], values[n]);                    O(logn)  
}  
           ↑           ↑  
        O(1)        O(1)
```

**Svar:**  $O(n) * O(\log n) = O(n \cdot \log n)$

# Repetition: vad blir tidskomplexiteten?

- Givet att den här koden från förra sliden redan körts:

```
int[] keys = { 1, 2, 3, 4, 5 };  
String[] values = { "One", "Two", "Three", "Four", "Five" };  
  
TreeMap<Integer, String> map = new TreeMap<>();  
  
for (int n : keys)  
{  
    map.put(keys[n], values[n]);  
}
```

...Vad blir då tidskomplexiteten om vi därefter gör följande operation?

```
String string = map.get(1);
```

- Operationen blir  $O(\log n)$  eftersom vi bara hämtar ut ett värde, och en TreeMap har  $O(\log n)$  på både insättning och uthämtning

# Iteratorer och Iterable

- Iterable är ett interface i Java lokaliserat i `java.lang` som bara innehåller en enda metod:

```
public interface Iterable<T>
{
    Iterator<T> iterator();
}
```

- Alla klasser som implementerar det här är en Iterable, vilket innebär att de kan loopas genom med en enhanced for-loop.
- En Iterator är själva objektet som utför iterationen. Beskrivs också av ett interface som man måste implementera i en klass:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove();
}
```

- Alla standardsamlingar i Java har redan sådana här implementerade



# Exempel: en itererbar klass

```
public class NumberRange implements Iterable<Integer>
{
    private final int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    public Iterator<Integer> iterator()
    {
        return new NumberIterator();
    }
}
```

```
private class NumberIterator implements Iterator<Integer>
{
    private int index = 0;

    public boolean hasNext()
    {
        return index < numbers.length;
    }

    public Integer next()
    {
        int value = numbers[index];
        index++;
        return value;
    }
}
```

← Nästlad inre klass

# Exempel: en itererbar klass

- När vi har både en iterable och en iterator definierade kan vi nu loopa genom objekt av vår klass med en enhanced for-loop:

```
NumberRange range = new NumberRange();  
  
for (int n : range)  
{  
    System.out.println(n);  
}
```

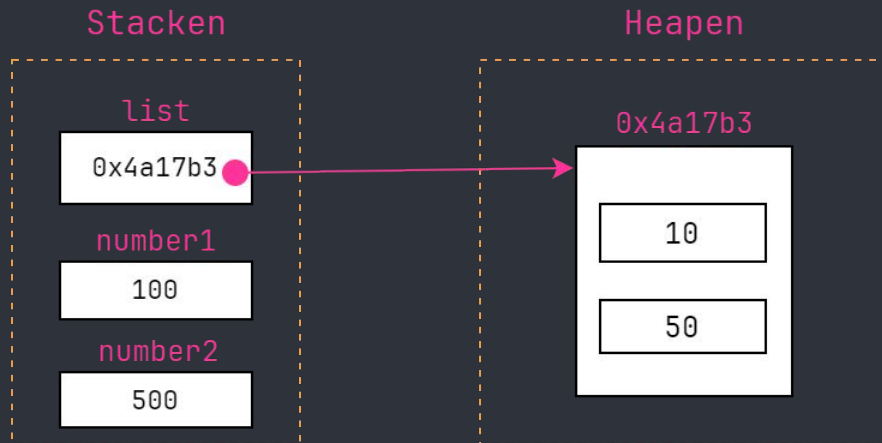
- Inte något som är viktigt att känna till på den här kursen, men **kan vara bra att veta om man vill ha en lite djupare förståelse** för vad enhanced for-loop egentligen gör
- **Inte vanligt att skriva sina egna iteratorer:** vi använder nästan alltid samlingar som redan har dem definierade (ArrayList, LinkedList, HashSet, etc)

# Snabb repetition: stack och heap

- I Java är alla objektvariabler egentligen **referenser** som pekar mot data som finns på **heapen**
- När vi t.ex. skapar en **ArrayList** så lagras den på heapen, medan **referensen** till den hålls på stacken
- Varje gång vi använder **new** heap-allokerar vi i Java. Primitiva typer såsom **integers** sparas däremot på stacken

## Programkod:

```
ArrayList<Integer> list = new ArrayList<>();  
  
list.add(10);  
list.add(50);  
  
int number1 = 100;  
int number2 = 500;
```



# Initialisera alltid variabler via konstruktorn

- Eftersom **null är defaulttillstånd** för alla objekt som deklarerats men ännu inte initialiserats är det viktigt att vi **inte lämnar några referenser “hängande”**
- Ett konstruerat objekt **bör alltid** vara i ett giltigt tillstånd (**state**). Om det t.ex. kan producera NullPointerExceptions när vi anropar metoder på objektet är det i ett ogiltigt tillstånd.
- **Vi har en konstruktor** för att hjälpa oss med detta. Namnet refererar till att den **alltid anropas vid konstruktion** av en instans: definierar vi ingen själv kommer en osynlig konstruktor som inte gör någonting att köras automatiskt i bakgrunden

# Problem: Tvåstegsinitalisering

```
public class User
{
    private String name;

    public void setName(String name)
    {
        this.name = name;
    }

    public void printName()
    {
        System.out.println("User: " + name.toUpperCase());
    }
}
```

- Vi försöker skapa ett objekt och använda det någonstans:

```
User user = new User();
user.printName();
```

//**NullPointerException** eftersom vi inte tilldelat användaren något namn ännu!

# Lösning: använd konstruktorn

```
public class User
{
    private String name;

    public void User(String name)    //Konstruktor
    {
        this.name = name;
    }

    public void printName()
    {
        System.out.println("User: " + name.toUpperCase());
    }
}
```

- Vi tvingas initialisera namnet direkt när vi skapar objektet i stället:

```
User user = new User("Dale Cooper");
user.printName();    //Ok!
```

# Märk alla instansvariabler private

- **private** ser till att **ingen utomstående kan ändra** dina variabler: de kan inte skriva över strängar, ta bort objekt ur samlingar och listor, och-så-vidare
- De kan **inte ens se** vilka variabler objektet innehåller. Det här uppfyller **principen om inkapsling** som är grundläggande för objekt-orientering

```
class Person
{
    private String name = "Greta Garbo";
}
```

I main():

```
Person person = new Person();
person.name = "Betty Davies";           // Kompileringsfel: Omöjlig ändring
```

# Märk variabler med final om möjligt

- **final** är ett nyckelord i Java som innebär att en variabel **inte kan förändras** när den väl har tilldelats ett värde (motsv. till **const** i andra språk som C, C++ och Javascript)
- Använd alltid när ni vet att någonting inte bör skrivas över: det kommunicerar tydlighet till andra utvecklare och hindrar busig kod från att kompileras

```
final String[] busterKeatonMovies = {"Sherlock Jr", "The General", "Steamboat  
Bill Jr", "The Cameraman"};
```

```
busterKeatonMovies[1] = "Hamburger: The Motion Picture";    //Kompileringsfel!
```



# Föredra enhanced for-loopar över for-loopar

- Enhanced for-loop, ibland kallad för både **“förenklad for-loop”** och **“avancerad for-loop”** eftersom man ogillar tydlighet i Java, **bör alltid användas** om man itererar över samlingar utan att ändra deras Innehåll

```
for (int num : list)
{
    doSomething(i);
}
```

**Vanliga for-loopar är ofta problematiska eftersom de:**

- **Producerar undantagsfel** av typen **IndexOutOfBoundsException** om vi gör ett misstag när vi ställer upp loopvillkoret
- **Är ineffektiva** när vi itererar över samlingar
- **Inte kan iterera över** många av de vanligt förekommande data-strukturerna som t.ex kartor och set, som saknar direkt indexering

# Problem: IndexOutOfBounds

**Dåligt exempel: Vi hamnar utanför samlingen vi försöker iterera över**

```
String[] names = {"James Kirk", "Jean-Luc Picard", "Benjamin Sisko",  
                  "Kathryn Janeway"};  
  
for (int i = 0; i <= names.length; i++)           //Fel jämförelseoperator!  
{  
    System.out.println(names[i]);  
}
```

- Lätt att skriva loopvillkoret fel och hamna out of bounds
- Behöver hålla reda på hur vi ska få längden. Java är inte konsekvent i sin namnsättning: arrayer har `length`, listor har `size()`, strängar har `length()`, streams har `count()`...
- Gör det svårt att generalisera kod med generics

# Lösning: enhanced for-loop

**Bra lösning: Vi kan aldrig hamna out of bounds med en foreach-loop!**

```
for (String name : names)
{
    System.out.println(name);
}
```

- Loopen går bara så länge det finns värden att loopa genom
- Ökad läsbarhet: koden är kort och ren!
- Vi behöver inte skriva loopvillkor manuellt
- Vi kan skicka både en array och en lista till en metod och loopa genom den med samma logik

# Problem: ingen indexering

Dåligt exempel: Den här koden kompilerar inte

```
TreeMap<String, Integer> scores = new TreeMap();
scores.put("Buster Keaton", 90);
scores.put("Fatty Arbuckle", 85);
scores.put("Harold Lloyd", 88);

for (int i = 0; i < scores.size(); i++)
{
    System.out.println(scores[i]); // Kompileringsfel: Maps saknar indexering
}
```

- Kartor har **inte indexbaserad åtkomst** för värden, till skillnad från en array eller en lista, eftersom det är en **helt annan sorts datastruktur** som i stället har nyckelbaserad åtkomst (key-based lookup)
- Det här innebär att vi **inte kan iterera över** dem på vanligt vis med en for-loop

# Ineffektiv lösning: använd en lista

## Dålig lösning: använd en mellanliggande lista

```
TreeMap<String, Integer> scores = new TreeMap();
scores.put("Buster Keaton", 90);
scores.put("Fatty Arbuckle", 85);
scores.put("Harold Lloyd", 88);

List<String> keys = new ArrayList<>(scores.keySet()); //Iterator stoppar in i listan!

for (int i = 0; i < keys.size(); i++)
{
    System.out.println(keys.get(i) + " scored " + scores.get(keys.get(i)));
}
```

- Skapar en lista först att lägga till strängarna i. Dålig lösning om det rör sig om stora samlingar: **tar  $O(n)$  i tid** och kräver **mycket extra minnes-resurser** (dålig platskomplexitet)
- **Behöver två get()-anrop**: ett för nycklarna och ett för värdena

# Bra lösning: enhanced for-loop

**Best Practice: använd en enhanced for-loop som kan iterera över ett Entry-set direkt!**

```
for (Map.Entry<String, Integer> entry : scores.entrySet())  
{  
    System.out.println(entry.getKey() + " scored " + entry.getValue());  
}
```

- Effektiv lösning: Behöver inte anropa get() för både nycklar och värden eftersom de redan finns i entrySet(), så uthämtning är  $O(1)$
- Clean och kort kod!
- Minneseffektiv: ingen extra lista
- Eliminerar  $O(n)$  för att kopiera över alla nycklar



# När vi inte ska använda enhanced for

- När vi **behöver ta bort något** från en samling medan vi itererar genom den. Enhanced for-loop kan inte ta bort element på ett säkert vis och kommer att kasta ett **ConcurrentModification-Exception**:

```
for (String name : names)
{
    if (name.equals("James Kirk"))
    {
        names.remove(name);          //Kastar undantagsfel
    }
}
```



# Använd Javas inbyggda API

- Vi har gått genom hur man skapar strukturer såsom stackar, köer, binära träd, osv för att få en bättre förståelse för hur de fungerar
- I verkligheten **bygger man dock sällan** sina egna ADT:s, på samma vis som man sällan skriver sina egna sorterings- eller krypteringsalgoritmer: man **använder beprövade bibliotek** i stället eftersom de är bättre optimerade än något vi kan göra för hand
- **Gör livet lättare genom att använda de inbyggda alternativ som finns**
- Java har ett **standardbibliotek** fullt av algoritmer och datastrukturer redan. Googla/fråga en LLM om råd ifall ni letar efter något specifikt som inte finns där (t.ex. grafer)
- Ofta vill man att de ska vara **“trådsäkra” (thread-safe)** så att de fungerar i flertrådade program

# java.util har all sortering ni behöver

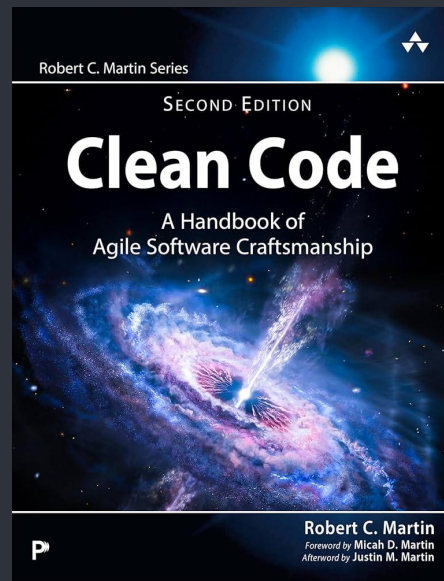
- **Arrays.sort()** använder **MergeSort** för att sortera objekt, men implementationen är egentligen en optimerad algoritm som heter TimSort
- TimSort är en hybrid av MergeSort och InsertionSort. Om en samling är liten eller delvis sorterad kommer den köra InsertionSort; om den är stor och/eller mestadels osorterad kommer MergeSort köras
- **QuickSort** används av Arrays.sort() för att **sortera primitiva datatyper** (int, double, osv). Java har en optimerad algoritm kallad för Double-Pivot QuickSort som används
- **Collections.sort()** använder TimSort för att sortera listor: kraftfull, effektiv och stabil sortering
- Om man vill sortera listor som innehåller objekt av egenskapade klasser behöver de klasserna implementera Comparable<T> eller Comparator<T> först

# Försök skriva så läsbar kod som möjligt

- Bra skriven kod är lätt att läsa och följa
- Den använder etablerade lösningar för generella (dvs vanliga) problem

## Saker som ofta är en code smell:

- Lång och krånglig kod som gör samma sak som kort och ren kod, fast sämre
- Långa if/else if/else-satser
- Långa metoder som försöker göra mer än en sak samtidigt: om man behöver skriva en förklarande kommentar mitt i en metod är det ofta ett tecken på att den bör delas upp



# Problem: hitta maxvärde i en samling

Problematisks lösning: använd en hemmasnickrad metod

```
public int findMax(List<Integer> list)
{
    int max = Integer.MIN_VALUE;

    for (int num : list)
    {
        if (num > max)
        {
            max = num;
        }
    }

    return max;
}
```

- Misslyckas om listan innehåller negativa tal
- Returnerar `Integer.MIN_VALUE` om listan är tom (-2147483648)
- Går sönder om list är null

# Bättre lösning: Collections.max()

```
int max = Collections.max(list);
```

- Kompilerar inte om list inte initialiserats
- Kastar undantag om listan är tom (och berättar att den gör det!)
- Optimerad för olika typer av samlingar (List, Set, Map)
- Support för comparators för objektjämförelser
- Finns även en min-funktion: `int min = Collections.min(list);`
- Lätt att utöka med customiserad logik:

```
String longest = Collections.max(words, Comparator.comparing(String::length));
```



# Ternaryoperatören, aka Elvisoperatören ?:

- En ternaryoperator är egentligen bara syntaktiskt socker som ersätter en if-else-sats med en enda rad instruktioner
- "Ternary" på engelska betyder tredelat; att något **består av tre delar**
- **Är (Uttryck) sant ? Gör då det här : Annars gör detta**

Exempel:

```
public boolean isPositive(int n)
{
    return n <= 0 ? false : true;
}
```



# Ternary i stället för korta if/else

**Detta:**

```
String status;  
  
if (score >= 50)  
{  
    status = "Pass";  
}  
else  
{  
    status = "Fail";  
}
```

**... kan ersättas med en enda rad:**

```
String status = score >= 50 ? "Pass" : "Fail";
```

- Kan ofta göra kod mer läsbar om man har en mängd if-else-satser

# Ternary i stället för else if

```
public String getGrade(int score)
{
    if (score >= 90)
    {
        return "A";
    }
    else if (score >= 80)
    {
        return "B";
    }
    else if (score >= 70)
    {
        return "C";
    }
    else
    {
        return "F";
    }
}
```

Ternary-operatoren kan returnera mer än en av två saker:

```
public String getGrade(int score)
{
    return score >= 90 ? "A"
        : score >= 80 ? "B"
        : score >= 70 ? "C"
        : "F";
}
```

- Tänk dock på: kedjade instruktioner blir mindre läsbara efter ett tag
- Bättre att använda switch, eller kanske enums i stället, för långa villkors-checkar



kaffepaus(15);



# Undantag (Exceptions)

- **Undantag** är någonting **oförutsett som sker** i ett program och avbryter det normala programflödet: ett **felobjekt kastas** ("throws") av programmet (Java Runtime Environment) och **måste "fångas" av koden** om man inte vill att de ska propagera genom programmet och krascha det
- Kan ha många orsaker: **nullpointers, otillgängliga filer,** någon dåre har försökt **dela ett tal med noll**, osv
- Vi tänker ofta på dem som någonting som är dåligt, men **undantag i sig är en fundamentalt bra sak** med programmering som hjälper oss att upptäcka och hantera fel på ett lite mer produktivt vis än att programmet bara kraschar
- Därför är det också **viktigt att veta hur de fungerar** och när och var man ska hantera dem

# Två sorters undantag: checked och unchecked

- **Märkta undantag (checked exceptions)** är sådana som kompilatorn tvingas oss att hantera innan programmet får kompileras
- **Antingen** behöver vi **fånga dem direkt** i metoden som genererar dem, **eller** också behöver vi **markera att metoden kan kasta undantag**: det blir då anroparens ansvar att fånga och hantera dem
- **Omärkta undantag (unchecked exceptions)** är svårare att ha att göra med eftersom vi ofta inte vet om att de finns: Java **kräver inte** att vi hanterar dem eller meddelar att de kan kastas

# Vanlig lösning: Hantera checked exception med try-with-resources

```
public class FileHandler
{
    public void readFile()
    {
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath)))
        {
            String line;
            while ((line = reader.readLine()) != null)
            {
                System.out.println(line);
            }
        }
        catch (IOException e)
        {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
}
```

- Ok kod! Vi ser till att fånga ett potentiellt undantag direkt. Nackdelen är att anroparen inte får bestämma hur det ska hanteras

# Bättre lösning: låt anroparen avgöra

```
public class FileHandler
{
    /**
     * Reads all lines from a text file
     * @throws IOException Thrown if file can't be accessed
     */
    public void readFile() throws IOException //Markera att metoden kan kasta
    {
        BufferedReader reader = new BufferedReader(new FileReader(filePath));
        while ((line = reader.readLine()) != null)
        {
            System.out.println(line);
        }
    }
}
```

- I stället för att logga ett fel till konsolen meddelar vi att metoden kan kasta, och **anroparen får bestämma själv** hur undantaget ska hanteras: det kanske bör loggas till databas, eller så kanske man vill att det lindar upp stacken och kraschar applikationen
- Vi gör också en markering i JavaDoc med **@throws-taggen**

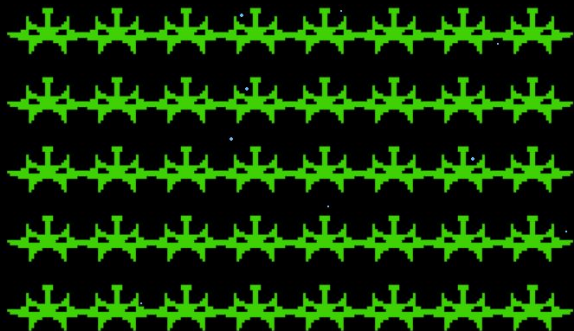
# Stack unwinding

- När ett undantag kastas **kommer det att propagera tillbaka genom callstacken** och poppa stack frame efter stack frame tills det stöter på ett catch-block som är menat att hantera det
- Det här kallas för **“stack unwinding”** på engelska: undantaget “lindrar upp” eller “spolar tillbaka” stacken genom att poppa lagren
- Ofta är det här **ett önskvärt beteende** från undantag: när de poppar stack frames på sin färd tillbaka genom stacken **frigörs minnesresurserna** på heapen och man undviker på så vis läckor
- **Stack unwinding** är viktig för att kunna “städa upp” i minnet när ett fel inträffar

# Exempel: Space Invaders

Score: 0

Lives: 3



# Ett exception lindar upp stacken

```
Microsoft Visual Studio Debug Console
INFO: FILEIO: [./Assets/Ship1.png] File loaded successfully
INFO: IMAGE: Data loaded successfully (352x352 | R8G8B8A8 | 1 mipmaps)
INFO: TEXTURE: [ID 3] Texture loaded successfully (352x352 | R8G8B8A8 | 1 mipmaps)
INFO: FILEIO: [./Assets/Barrier.png] File loaded successfully
INFO: IMAGE: Data loaded successfully (704x704 | R8G8B8A8 | 1 mipmaps)
INFO: TEXTURE: [ID 4] Texture loaded successfully (704x704 | R8G8B8A8 | 1 mipmaps)
Wall texture loaded successfully.
WARNING: FILEIO: [./Assets/NotAlien.png] Failed to open file
Wall instance destroyed. Count: 4, ID: 000001E89F4A4750
Current instance count: 4
Wall instance destroyed. Count: 3, ID: 000001E89F4A4764
Current instance count: 3
Wall instance destroyed. Count: 2, ID: 000001E89F4A4778
Current instance count: 2
Wall instance destroyed. Count: 1, ID: 000001E89F4A478C
Current instance count: 1
Wall instance destroyed. Count: 0, ID: 000001E89F4A47A0
Current instance count: 0
INFO: TEXTURE: [ID 4] Unloaded texture data from VRAM (GPU)
Wall texture unloaded.
INFO: TEXTURE: [ID 3] Unloaded texture data from VRAM (GPU)
INFO: TEXTURE: [ID 2] Unloaded texture data from VRAM (GPU)
INFO: SHADER: [ID 3] Default shader unloaded successfully
INFO: TEXTURE: [ID 1] Default texture unloaded successfully
INFO: Window closed successfully
Error loading texture: Failed to load ./Assets/NotAlien.png
```

- Vi kan bokstavligen talat se hur stacken lindas upp
- Ett TextureLoadingException smäller, och vi kan se hur vägg efter vägg raderas när undantaget börjar poppa stacken, och hur texturerna till sist läses ur från GPU:n
- Undantaget fångas i main() och loggas, efter att det poppat hela stacken och städat upp efter sig

```
C:\Users\46705\Desktop\HT 2024\API-design och best practises\Final_Assignment\SpaceInvaders_2023\SpaceInvaders_2023\Build\Debug\Game.exe (process 22292) exited with code 2 (0x2).
Press any key to close this window . . .
```



# Dålig praxis: tomma eller generella catchblock

- Om ett undantag är viktigt nog att kasta så är det också viktigt nog att hantera på något vis: **stoppa aldrig in tomma catch-satser** som fångar ett generellt Exception bara för att “safeguarda” koden ifall något oförutsett skulle ske någonstans
- **Catch-block är menade att vara specifika:** vi vill ha information om vad det är som går fel så att vi kan debugga vår kod
- **Vi vill också vara tydliga** med vad vi **förväntar oss** för undantag: ett block som bara fångar ett “Exception e” kommer att **snappa upp alla undantag** som propagerar genom callstacken, inkluderat sådana som vi kanske vill hantera på något annat ställe

# Dålig praxis: tomma eller generella catchblock

## Dåligt exempel:

```
System.out.println("Enter a number: ");
String userInput = scanner.nextLine();

try
{
    int number = Integer.parseInt(userInput);
    System.out.println("You entered: " + number);
}
catch (Exception e) { }
```

- Kommunikerar inte tydligt vad för problem som förväntas
- Loggar inget, så vi kommer aldrig få veta att något fel inträffat
- Catch-blocket fångar ett generellt "Exception" och kommer då att fånga **ALLA** sorters undantag: om något kastas tidigare i programmet och inte hanterats än kommer det att sväljas här

# Bra praxis: specifik felhantering

```
System.out.println("Enter a number: ");
String userInput = scanner.nextLine();

boolean done = false;
while(!done)
{
    try
    {
        int number = Integer.parseInt(userInput);
        System.out.println("You entered: " + number);
        done = true;
    }
    catch (NumberFormatException e)
    {
        System.out.println("Integer couldn't be parsed!");
    }
}
```

- Ett specifikt fel fångas
- Specifik felinformation loggas
- Vi wrapper även vår try-catch i en loop så att **number** aldrig kan lämnas i ett oinitialiserat tillstånd och skapa andra problem senare

# Logga eller kasta undantag, men kasta inte om dem

- Regel 1: Om man kastar ett undantag manuellt bör det också fångas någonstans
- Regel 2: Om man fångar ett undantag ska det hanteras och inte kastas om

- Här föreslår JetBrains det exakt motsatta, men det är ett dåligt råd!
- Ett undantag bör bara fångas en gång och loggas en gång eftersom vi vill att det ska vara spårbart

## Logging the exception and rethrowing it

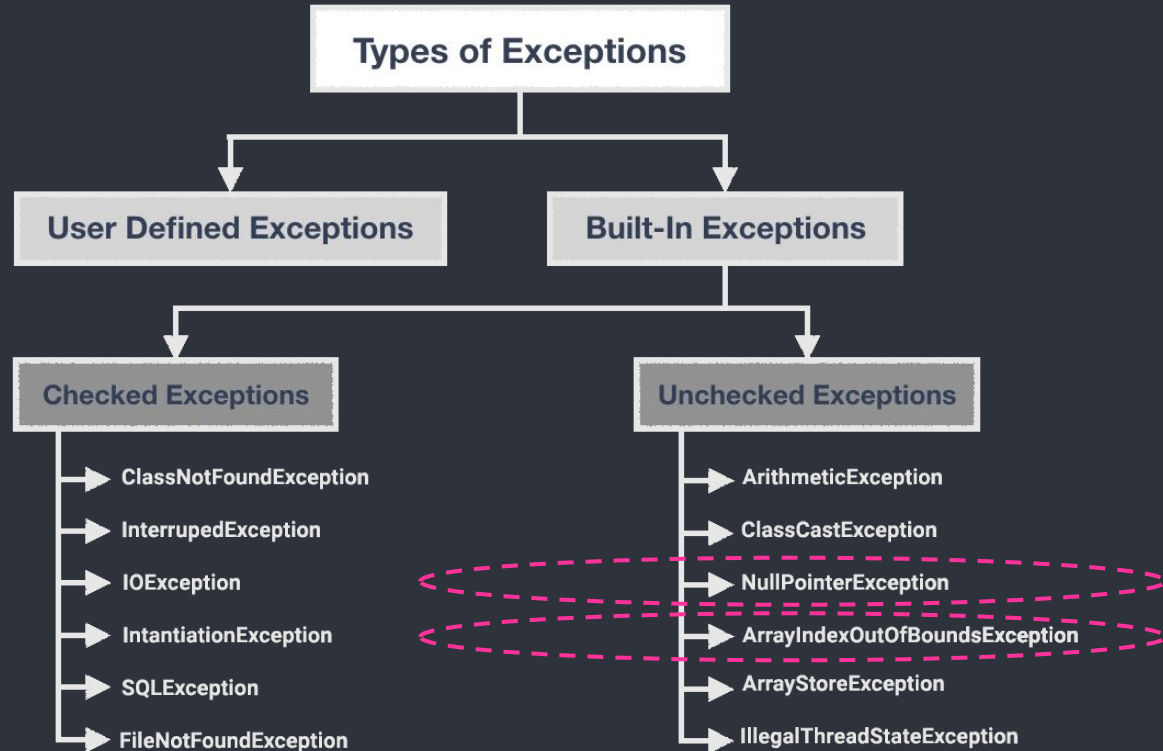
In an ideal world, the catch block would identify `IOException`, then print an error message to the console and rethrow the exception to handle it later.

This way, you get the full picture of what went wrong.

```
try {  
    int division = 10 / 0;  
} catch (ArithmeticException ex) {  
    System.out.println("An exception occurred: " + ex.getMessage());  
    throw ex;  
}
```

# Omärkta (unchecked) undantag

- Av de omärkta undantag som finns är det främst två som vi brukar råka ut för
- Dessa uppstår under körtid och oftast på grund av programmeringsfel
- Ibland kan vi ta höjd för dem med try-catch-block och liknande, men vi introducerar då nya problem för anroparen



# Följ best practices för att undvika unchecked exceptions så gott det går

- Initialisera alltid variabler i konstruktorn
- Validera alltid input, både från användaren och när något skrivs till en list från en fil, databas, etc
- Se till att aldrig returnera null om möjligt:

```
public List<String> getNames()
{
    return names != null ? names : Collections.emptyList();
}
```

- Använd foreach-loopar i stället för for-loopar för att förebygga problem som uppstår när man itererar över samlingar
- Om man vet att en viss bit kod kan producera omärkta undantag är det helt ok att antingen förebygga detta med en try-catch eller att märka metoden så att andra anropare också blir medvetna om det

# Man kan skriva sina egna exceptionklasser

- Att **skriva egna undantag** kan underlätta felhantering och debugging och göra koden både **tydligare och mer robust**
- Java har redan en mängd undantag inbyggda, så egna undantag **bör bara skapas när de är meningsfulla** att använda
- Att kasta samma standardundantag överallt gör dock debugging svårare. Att generera ett **InvalidUserInputException** i stället för ett **IllegalArgumentException** kan göra koden **lättare att felsöka**
- Ett finansiellt system kanske vill kasta ett **InsufficientFundsException** i stället för ett generellt Exception om det inte går att dra ett värde från en summa, osv

# Exempel på custom exception

- Man skapar egna undantag genom att ärva från Javas Exception-klass:

```
class InvalidAgeException extends Exception
{
    public InvalidAgeException(String message)
    {
        super(message);
    }
}
```

- “Extends” innebär att man ärver från en annan klass i Java
- “super” innebär att man skickar vidare anropet till basklassen, också kallad för superklassen
- Vårt custom exception fungerar med andra ord som en wrapper som delegerar vidare till Exception-klassen



# Exempel på hur vårt custom exception kan användas

```
public static void registerUser(int age) throws InvalidAgeException
{
    if (age < 18)
    {
        throw new InvalidAgeException("Age must be 18 or above to register.");
    }

    System.out.println("User registered successfully.");
}
```

```
public static void main(String[] args)
{
    try
    {
        registerUser(15);
    }
    catch (InvalidAgeException e)
    {
        System.out.println("Registration failed: " + e.getMessage());
    }
}
```

# [Kodexempel]

Koden finns i paketet `CustomException`

# Problem kan ofta omformuleras

- Vi gjorde en `multiplyAll` i Laboration 1:

I metoden `multiplyAll` ska alla värde multiplicera med alla andra värden i en lista. Om listan innehåller värdena { 0, 1, 2, 3 } så ska algoritmen multiplicera och summera enligt följande:

```
{ 0, 1, 2, 3 } -> 0 * 1 (0) +  
{ 0, 1, 2, 3 } -> 0 * 2 (0) +  
{ 0, 1, 2, 3 } -> 0 * 3 (0) +  
{ 0, 1, 2, 3 } -> 1 * 2 (2) +  
{ 0, 1, 2, 3 } -> 1 * 3 (3) +  
{ 0, 1, 2, 3 } -> 2 * 3 (6)
```

Med listan { 0, 1, 2, 3 } kommer alltså summan bli 11.

- För varje gång en loop körs ska ett tal multipliceras med alla tal framför sig, men inte de bakom sig

# Problem kan ofta omformuleras

- Man gör normalt en lösning som såg ut så här i Labb 1:

```
public long multiplyAll(List<Integer> numberList)
{
    long sum = 0;

    for (int i = 0; i < numberList.size(); i++)
    {
        for (int j = i + 1; j < numberList.size(); j++)
        {
            sum += (long) numberList.get(i) * numberList.get(j);
        }
    }

    return sum;
}
```

- **Giltig lösning!** Ser ut som det enklaste och bästa sättet att lösa uppgiften så som den är formulerad
- Tidskomplexiteten är kvadratisk,  **$O(n^2)$**

# Optimerad lösning: $O(n)$ !!

```
public long multiplyAll(List<Integer> numberList)
{
    long sumOfAllElements = 0;
    long sumOfAllSquares = 0;

    for (int num : numberList)
    {
        sumOfAllElements += num;
        sumOfAllSquares += (long) num * (long) num;
    }

    return (sumOfAllElements * sumOfAllElements - sumOfAllSquares) / 2;
}
```

- Renare kod och bättre tidskomplexitet
- Lätt att koda, svårare att klura ut

# Optimerad lösning: förklaring

- Vi omformulerar problemet matematiskt:

$(\text{summanAvAllaTal} * \text{summanAvAllaTal}) - \text{talSomMultipliceratsMedSigSjälvt}$

---

2

- $(\text{summanAvAllaTal} * \text{summanAvAllaTal})$  ger oss alla multiplikationer som är möjliga - inklusive självmultiplikationer som vi vill undvika (såsom  $1 \times 1$ ,  $2 \times 2$ , osv)
- $\text{talSomMultipliceratsMedSigSjälvt}$  tar bort dessa självmultiplikeringar
- Vi dividerar sedan med 2 för att ta **kompensera för de tillfällena där vi dubbelräknar** varje par (exempel:  $1 \times 2$  och  $2 \times 1$ )

# Optimerad lösning

- Oftast **inte vår uppgift** som utvecklare att omformulera problem
- Vi får en specifikation och så säger någon "Bygg det här, genast!"
- Upp till någon **ingenjör eller matematiker** eller annan typ av person som hatar sig själv att optimera formeln vi använder för beräkningar (tack och lov!)
- Kan dock vara **värt att reflektera över om vi har den mest effektiva problembeskrivningen** eller ej: ofta är det sådana här saker som avgör effektiviteten för beräkningar och instruktioner
- **Tidskomplexitet kan vara en bra indikator:** Är den  $O(n^2)$  eller högre kan man fundera på om det finns bättre sätt att tackla problemet

# Sammanfattning av förmiddagen

## Konstruktorer

- Är till för att initialisera variabler
- Gör att vi inte riskerar nullpointers
- Undviker tvåstegsinitialisering

## foreach-loopar

- Bästa valet för att iterera över samlingar
- Säkra, effektiva, renare kod

## Javas inbyggda API

- All sortering man behöver
- Inbyggda metoder är mer effektiva än hemmasnickrade lösningar

## Undantag och felhantering

- Följ best practices för att undvika unchecked exceptions
- Checkad och unchecked exceptions
- Lindar upp stacken om de inte fångas