Tematyka ćwiczenia:

- Wyjątki- składnia, tworzenie, mechanizm, obsługa
- Hierarchia wyjątków – Error, Exception, RuntimeException
- Blok try-catch
- Blok finally

wyjątek – mechanizm kontroli przepływu służący do obsługi zdarzeń wyjątkowych (w szczególności błędów)

1.1 Wyjątki składnia

```
try {
	// monitorowany blok kodu
}

catch (ExceptionType1 exOB)
{
	// to się wykona, jeżeli zostanie zgłoszony
	// wyjątek typu ExceptionType1
}

catch (ExceptionType2 exOB)
{
	// to się wykona, jeżeli zostanie zgłoszony
	// wyjątek typu ExceptionType2
}

finally
{
	// kod wykonywany po zakończeniu bloków
	// try i catch; służy do zwalniania zasobów
{
```

1.2 Generowanie nowego wyjątku

```
if (o == null)
throw new NullPointerException();
```

1.3 Tworzenie - rozszerzanie interfejsu klasy bazowej

```
class MyException extends Exception { }
```

1.4 Nieobsłużone wyjątki

Metoda *głebiej()* w pokazanej poniżej klasie *NieobsługiwanyWyjatek* zgłasza wyjątek klasy *Exception* (podstawowa klasa reprezentująca wyjątki w Javie), gdy przekazany jej parametr ma wartość **null**.

```
public class NieobsługiwanyWyjatek
{
	void głebiej(String s) throws Exception
	{
		System.out.println("poczatek głebiej");
		if (s == null) throw new Exception();
		System.out.println("koniec głebiej");
	}
```

```
        void głeboko(String s) throws Exception
        {
                System.out.println("poczatek głeboko");
                głebiej(s);
                System.out.println("koniec głeboko");
        }

        public static void main(String[] args) throws Exception
        {
                NieobsługiwanyWyjatek ue = new NieobsługiwanyWyjatek();
                System.out.println("przed głeboko");
                ue.głeboko(null);
                System.out.println("po głeboko");
        }
}
```

Z powodu zgłoszenia wyjątku stos wywołań jest rozwijany, a ponieważ żadna z kolejno znajdujących się tam metod *głębiej()*, *głęboko()* i *main()* nie zawiera kodu obsługi wyjątku, program zostaje przerwany. W takiej sytuacji Java wypisuje na standardowe wyjście błędu informacje o wyjątku, który spowodował przerwanie programu. Ich poziom szczegółowości zależy od ustawień kompilatora, ale przy ustawieniach domyślnych podawany jest ślad stosu wywołań z chwili wystąpienia wyjątku oraz numery linii w kodzie źródłowym odpowiadające kolejnym jego pozycjom.

1.5 Kontrolowanie obsługi wyjątków przez kompilator

Warto zwrócić uwagę, że w deklaracjach wszystkich metod klasy *NieobsługiwanyWyjątek* występuje klauzula **throws**, która informuje, jakich wyjątków można się spodziewać w wyniku wywołania danej metody. W tym wypadku klauzula ta jest wymuszona przez kompilator, gdyż z metody mogą wydostać sie nieobsłużone wyjątki. Dzięki temu programista nie przeoczy żadnego wyjątku i jeżeli nie chce wszystkich obsłużyć musi świadomie wymienić je (bądź ich nadklasy) w deklaracji metody. Jeżeli wymienianych jest kilka klas, ich nazwy oddziela się przecinkiem.

.

1.6 Obsługa wyjątków- blok Catch

Do obsługi wyjątków służy instrukcja **try-catch**. Po **try** podaje się blok instrukcji, których wyjątki chcemy obsługiwać. Następnie podawana jest lista bloków **catch**, które przypominają deklaracje metod. Każda z nich obsługuje wyjątki określonego typu. Typ wyjątku wraz z nazwą zmiennej, na którą zostanie przypisany łapany egzemplarz, podaje się między nawiasami bezpośrednio po słowie kluczowym **catch**. Do tej zmiennej można się odwoływać w kodzie obsługi wyjątku. Do obsługi wyjątku wybierany jest zawsze pierwszy pasujący blok **catch**. Ponieważ wyjątki są obiektami "pasowanie" oznacza tu po prostu możliwość przypisania wyjątku na zmienną zadeklarowaną po słowie kluczowym **catch**. Wszystkie dalsze bloki **catch** są pomijane, nawet jeżeli pasowały. Jeżeli wyjątek nie pasuje do żadnego bloku **catch**, nie zostaje obsłużony i stos wywołań jest nadal rozwijany. W pokazanej poniżej klasie *ObsługiwanyWyjątek*, która jest modyfikacją poprzedniego przykładu, wyjątek *Exception* jest już obsługiwany i nie powoduje przerwania programu.

```
public class ObsługiwanyWyjątek
{
  void głębiej(String s) throws Exception
  {
    System.out.println("początek głębiej");
    if (s == null) throw new Exception();
    System.out.println("koniec głębiej");
  }

  void głęboko(String s)
```

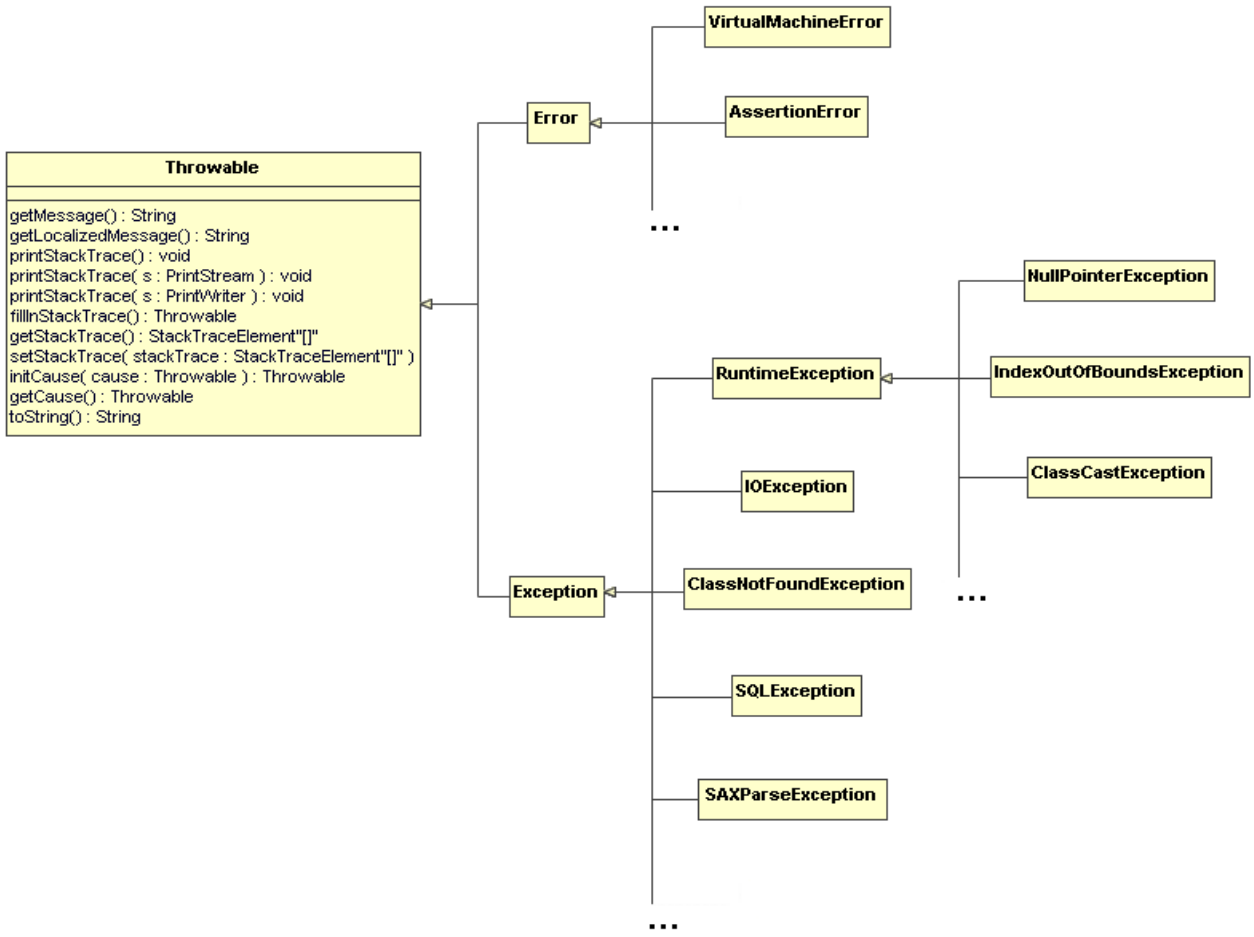
```
  {
    try
    {
      System.out.println("początek głęboko");
      głębiej(s);
      System.out.println("koniec głęboko");
    }

    catch (Exception e)
    {
      System.out.println("obsługa wyjątku");
      e.printStackTrace(System.out);
    }
    System.out.println("po obsłużeniu wyjątku");
}

  public static void main(String[] args)
  {
    ObsługiwanyWyjątek ep = new ObsługiwanyWyjątek();
    System.out.println("przed głęboko");
    ep.głęboko(null);
    System.out.println("po głęboko");
  }
}
```

Warto zwrócić uwagę, że metody *głęboko()* i *main()* nie muszą już posiadać klauzuli **throws**. Na konsolę nadal wypisywane są informacje o wyjątku, ale jest to spowodowane użyciem w kodzie obsługi wyjątku metody ***printStackTrace()*** z parametrem ***System.out***. Metoda ta jest odziedziczona z klasy ***Throwable***, po której muszą dziedziczyć wszystkie wyjątki.

2. Hierarchia wyjątków w Javie



2.1 Klasa *Error*

Jedyne dwie bezpośrednie podklasy ***Throwable*** to ***Error*** i ***Exception***. Wyjątki dziedziczące po ***Error*** reprezentują poważne problemy, których aplikacja nie będzie w stanie rozwiązać.

2.2 Klasa *Exception*

Wyjątki dziedziczące po ***Exception*** reprezentują sytuacje, na które dobrze napisana aplikacja powinna być przygotowana. To właśnie tę klasę rozszerza się tworząc własne rodzaje wyjątków. Jej przykładowe podklasy to:

IOException	reprezentuje sytuacje wyjątkowe związane z wejściem/wyjściem,
ClassNotFoundException	wskazuje, że maszyna wirtualna nie odnalazła klasy o nazwie podanej jako napis
SQLException	reprezentuje sytuacje wyjątkowe związane z dostępem do bazy danych
SAXParseException	wskazuje, że podczas parsowania dokumentu XML wystąpił błąd

2.3 Klasa *RuntimeException*

Bardzo ciekawą podklasą ***Exception*** jest ***RuntimeException***, która sama posiada wiele podklas. Wyjątki rozszerzające ***RuntimeException*** mogą wystąpić podczas typowych operacji, jak rzutowanie zmiennej, odwołanie się do elementu tablicy lub odwołanie się do składowej obiektu. Ich wystąpienie zazwyczaj oznacza, że programista popełnił błąd w swoim kodzie lub nieumiejętnie korzystał z kodu napisanego przez innych. Maszyna wirtualna wykrywa wystąpienie takich błędów w trakcie działania programu i informuje o tym, zgłaszając odpowiedni wyjątek.

3. Wyjątki kontrolowane/niekontrolowane

Mimo, że często używa się wyjątków ze standardowych bibliotek Javy, tworzenie nowych wyjątków nie jest niczym nadzwyczajnym. Tworząc nowe wyjątki zazwyczaj rozszerza się albo bezpośrednio klasę *Exception* albo *RuntimeException*.

- Wyjątki rozszerzające *Exception*, ale nie *RuntimeException*, nazywane są **wyjątkami kontrolowanymi** (*ang. checked exceptions*), gdyż kompilator pilnuje, aby programista ich nie przegapił i albo obsłużył albo wymienił w klauzuli **throws**.

- Wyjątki rozszerzające *RuntimeException* to **wyjątki niekontrolowane** (*ang. unchecked exceptions*) i ich kontrola ze strony kompilatora nie dotyczy. Oba rodzaje są potrzebne i nie należy o nich zapominać.

4. Zgłaszanie wyjątków w bloku catch

W reakcji na obsługiwany wyjątek, w bloku **catch** można zgłosić nowy wyjątek. Wyjątki zgłoszone w bloku **catch** nie są obsługiwane przez tę samą instrukcję **try-catch**.

➢ Ponowne zgłaszanie obsługiwanego wyjątku

Czasami obsługiwany wyjątek jest zgłaszany ponownie, gdyż w tym kontekście nie wiadomo jak go obsłużyć.

```
void głębiej() throws Exception
{
        throw new Exception();
```

```
}


void głęboko() throws Exception
{
        try
        {
                głębiej();
        }

        catch (Exception e)
        {
                throw e;
        }
}
```

Ponowne zgłoszenie nie zmienia zapamiętanego śladu wywołań. Nie jest on wypełniany w chwili zgłoszenia wyjątku, ale w chwili jego utworzenia. Odpowiada za to konstruktor z nadklasy *Throwable*, który wywołuje metodę *fillInStackTrace()*. Przy pomocy tej metody można samemu podmienić na aktualny ślad stosu wywołań w ponownie zgłaszanym wyjątku.

```
void głębiej() throws Exception
{
  throw new Exception();
}

void głęboko() throws Exception
{
  try
  {
    głębiej();
  }
  catch (Exception e)
  {
    e.fillInStackTrace();
    throw e;
  }
}
```

- **Wyjątek powodujący zgłoszenie innego wyjątku**

Od Javy 1.4 do zgłaszanego wyjątku można dołączyć wyjątek, który spowodował jego wystąpienie. Służy do tego metoda ***initCause(),*** która jako parametr przyjmuje dołączany wyjątek. Dla wygody do klas **Throwable**, **Exception** i **RuntimeException** dodano również po dwa nowe konstruktory. Pierwszy, przyjmujący jako jedyny parametr wyjątek do dołączenia oraz drugi, przyjmujący jako pierwszy parametr napis przenoszony przez wyjątek i jako drugi parametr wyjątek do dołączenia. Jeżeli mamy zamiar w ten sposób łączyć wyjątki w ciągi, warto dodać takie konstruktory do nowo definiowanych wyjątków.

```
class MójNowyWyjątek extends Exception
{
  public MójNowyWyjątek()
  {
    super();
  }

  public MójNowyWyjątek(String message)
  {
    super(message);
  }

  public MójNowyWyjątek(Throwable cause)
  {
```

```
    super(cause);
  }

  public MójNowyWyjątek(String message, Throwable cause)
  {
    super(message, cause);
  }
}

public class WyjątekSpowodowanyWyjątkiem
{
  public static void main(String[] args) throws MójNowyWyjątek
  {
    try
    {
      throw new Exception("To ja jestem wszystkiemu winien.");
    } catch (Exception e)
    {
      throw new MójNowyWyjątek("To nie moja wina.", e);
    }
  }
}
```

Informacje o dołączonym wyjątku można uzyskać przy pomocy metody ***getCause()*.** Informacje te są również wypisywane na standardowe wyjście błędu, jeżeli wykonanie programu zostało przerwane przez nieobsłużony wyjątek.

5. Blok finalny

Do instrukcji **try** można dodać blok **finally**. Zawarty w nim kod jest wykonywany zawsze, niezależnie od tego, czy w bloku **try** jest zgłaszany wyjątek czy nie i niezależnie od tego, czy **try** kończy się normalnie, z powodu **return**, czy z powodu **break** lub **continue**. Blok **finally** jest doskonałym miejscem na zwalnianie wszelkiego rodzaju zasobów. Umieszcza się go bezpośrednio po ostatnim bloku **catch**.

```
try {
  //kod który może zgłosić wyjątki
} catch (Typ1 w) {
  //obsługa wyjątków Typ1
} catch (Typ2 w) {
  //obsługa wyjątków Typ2
} catch (Typ3 w) {
  //obsługa wyjątków Typ3
} finally {
  //kod wykonywany niezależnie od wystąpienia wyjątku
  //tu zwalniamy zasoby
}
```

Można też użyć samego bloku **finally** i nie obsługiwać żadnych wyjątków.

```
try {
  //kod który może zgłosić wyjątki
} finally {
  //kod wykonywany niezależnie od wystąpienia wyjątku
  //tu zwalniamy zasoby
}
```

Jeżeli w bloku **try** nie wystąpił wyjątek, kod z bloku **finally** wykonywany jest bezpośrednio po jego zakończeniu. Jeżeli wyjątek wystąpił, ale nie pasuje do żadnego bloku **catch** lub nie umieszczono żadnego bloku **catch**, kod z bloku **finally**wykonywany jest bezpośrednio po wystąpieniu wyjątku, a przed rozpoczęciem rozwijania stosu wywołań w poszukiwaniu innej

instrukcji **try-catch**, która mogłaby ten wyjątek obsłużyć. W końcu, jeżeli wyjątek wystąpił i jest pasujący blok **catch**, to kod z bloku **finally** wykonywany jest bezpośrednio po zakończeniu obsługi wyjątku. Opisaną kolejność obrazuje poniższy przykład.

```
public class TestFinally
{
  public static void main(String[] args)
  {
    try
    {
      System.out.println("Zewnętrzne try");
      try
      {
        System.out.println("Pierwsze wewnętrzne try");
      }

      finally
      {
        System.out.println("Pierwsze wewnętrzne finally");
      }

      try
      {
        System.out.println("Drugie wewnętrzne try");
        throw new Exception();
      }

      finally
      {
        System.out.println("Drugie wewnętrzne finally");
      }
    }

    catch (Exception e)
    {
      System.out.println("Obsługa wyjątku");
    }

    finally
    {
      System.out.println("Zewnętrzne finally");
    }
  }

}
```

5.1 Zaginięcie wyjątku

Z użyciem **finally** wiąże się jeden mankament. Kod zawarty w tym bloku w żadnym wypadku nie powinien zgłaszać wyjątku. Jeżeli wyjątek zgłaszany w bloku **try** nie został obsłużony, a w **finally** również zgłaszany jest wyjątek, to wyjątek z bloku **try** zaginie. Tak właśnie dzieje się w poniższym przykładzie.

```
class WażnyWyjątek extends Exception {}
class JakiśInnyWyjątek extends Exception {}
class NieważnyWyjątek extends Exception {}

public class ZaginięcieWyjątku
{
  static void niebezpiecznyKod() throws WażnyWyjątek, JakiśInnyWyjątek
  {
    throw new WażnyWyjątek();
  }

  public static void main(String[] args) throws Exception
  {
    try
    {
      //tego wyjątku nie wolno przegapić
      niebezpiecznyKod();
```

```
    }

    catch (JakiśInnyWyjątek w)
    {
      //obsługa jakiegoś innego wyjątku
    }

    finally
    {
      //zwalnianie zasobów
      throw new NieważnyWyjątek();
    }
  }
}
```

********** **ZADANIE** **********

Wykonać aplikację odporną na błędy oraz:

- utworzyć własny wyjątek przechowujący opis oraz numer kodu błędu
- zdefiniować konstruktor oraz metody wyrzucające powyższy wyjątek
- zastosować blok obsługi wyjątków try/catch/finally

********** **PRZYKŁAD** **********

// **Klasa Zegarek**

```
public class Zegarek
{
            private int sekunda;
            private int minuta;
            private int godzina;

            public int ustawGodzine(int g)
            {
                  godzina=g;
                  return 1;
            }

            public int ustawMin(int g)
            {
                  minuta=g;
                  return 1;
            }

            public int ustawSek(int g)
            {
                  sekunda=g;
                  return 1;
            }

            public Zegarek(int godzina, int minuta, int sekunda)throws
MyException
            {
                  this.sekunda=sekunda;
                  this.minuta=minuta;
                  this.godzina=godzina;
                  try
                  {
                        if ((godzina>24)&& (minuta>60)&& (sekunda>60))
                        {
```

```
                        throw new MyException("2");
                  }
                  if ((godzina>24)&& (minuta>60)&& (sekunda<0))
                  {
                        throw new MyException("1");
                  }
            }

            catch (MyException e)
            {
                  System.out.println(e.getMessage());
            }



      }

      public String toString ()
      {
            String czas = new String();
            czas="h"+godzina+"m"+minuta+"s"+sekunda;
            return czas;
      }

      }
```

****

```
// klasa  MyException

public class MyException extends Exception
{
      public MyException(String s)
      {
            super(s);
      }
}
```

****

```
// klasa Test

public class Test
{
      public static void main(String[] args)throws MyException
      {
            Zegarek z = new Zegarek(0,0,0);
            System.out.println(z);
                  z.ustawGodzine(25);
                  z.ustawMin(65);
                  z.ustawSek(67);
                  System.out.println(z);
                  z.ustawGodzine(-2);
                  z.ustawMin(-4);
                  z.ustawSek(-5);
                  System.out.println(z);

      }
}
```