UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 07 – Tratamento de Exceções e Generics

1. EXCEÇÕES EM JAVA

Todo programa de computador deve prever situações anômalas, nas quais algo inesperado pode ocorrer, independentemente da linguagem de programação utilizada. Um exemplo típico de exceção é a entrada de dados incorreta, onde um usuário entra com um tipo de dado incompatível com o que era esperado – ex: digita uma frase quando deveria digitar um valor inteiro. Algumas linguagens oferecem pouco ou nenhum suporte para este tipo de problema, sendo responsabilidade do programador se precaver da melhor maneira possível utilizando estruturas condicionais e funções que se adequem ao seu desejo. Já outras linguagens, principalmente aquelas com tipagem fraca (tipos não são tão rígidos), são muito mais robustas neste sentido, mas podem acabar dificultando a descoberta de erros. Outros exemplos típicos de exceções são: tentativa de acesso a um índice fora dos limites, divisão por zero, estouro de memória, ou overflow numérico.

> Você já parou para pensar ou, quem sabe, até mesmo tentar implementar um programa em C que não gere um erro ao ler um caractere quando era esperado um número?

Algumas possíveis formas de tratar erros poderiam ser:

- Apenas desconsiderar a operação caso um valor inválido seja passado: isso gera uma incerteza sobre o sucesso ou não da operação realizada;
- Desconsiderar a operação e mostrar uma mensagem de erro: a classe fica atrelada à interface com o usuário escolhida (console ou interface gráfica?);
- Desconsiderar a operação e retornar um código de erro: obriga a realização de testes após a chamada do método para verificar se a operação teve efeito ou não, além de obrigar a criação de valores simbólicos de erro – os chamados números mágicos (ex: 1 indica erro A, 2 indica erro B, 3 indica erro C), que são considerados uma má prática.

Uma maneira mais elegante e efetiva de tratar exceções seria a criação de um conceito de exceção dentro da própria linguagem de programação, separando a parte de lógica de negócio do sistema, da parte de tratamento de erros. Essa separação facilita a legibilidade e manutenção do código. A ideia é que exceções possam interromper o fluxo normal do programa, podendo ser capturadas e tratadas no método onde ocorreram ou ser lançadas adiante e tratadas nos blocos de código que chamaram o método que gerou a exceção. Se nenhum método se responsabilizar pelo tratamento do erro, então o programa poderá ser terminado ou terá um evento cancelado (neste sentido já entraríamos no aspecto de orientação a eventos, além de conceitos sobre threads, que fazem parte da disciplina de sistemas operacionais).

A linguagem de programação Java utiliza um mecanismo de Exceções que pode ser utilizado para indicar condições anormais dentro de um programa sem fazer com que o mesmo encerre de

maneira abrupta. Esse mecanismo, na verdade, utiliza o mesmo pensamento de classes e objetos que vemos constantemente no paradigma de orientação a objetos, e nada mais é do que uma classe denominada de Throwable. É possível, por exemplo, criar uma instância de um objeto Throwable:

```
Throwable erro = new Throwable("Error");
```

Todo objeto que seja uma instância direta ou indireta de Throwable é um objeto de exceção que pode ser arremessado adiante, como o nome sugere, ou tratado de alguma maneira pelo programa. Este conceito de arremessar adiante merece atenção especial, mas trataremos dele daqui a pouco. Antes disso é interessante conhecer a hierarquia de classes de exceções, pois Throwable é apenas a superclasse pronta do Java, mas existem outras mais específicas. Essa hierarquia se divide em dois ramos: erros e exceções. O primeiro ramo descreve erros internos de execução da máquina virtual Java, que não podem ser tratados, ao passo que o segundo abrange exceções em tempo de execução ou não, que podem ser tratados. Para ilustrar esta hierarquia e apresentar alguns exemplos de exceções usuais do Java é apresentada a Figura 1.

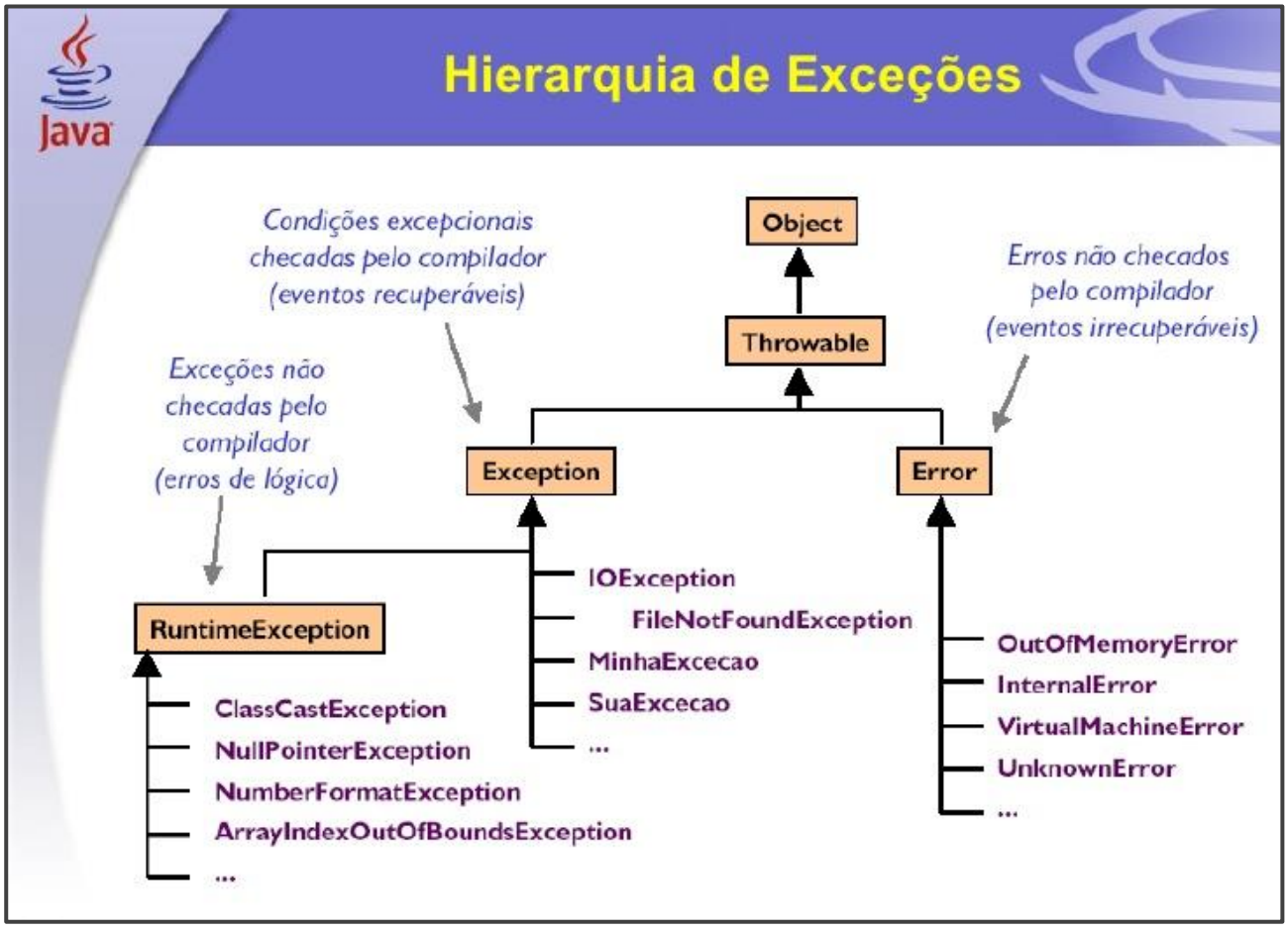


FIGURA 1 – Hierarquia de Exceções em Java (Fonte: Exceções – Regis Pires Magalhães)

O grupo de exceções da CLASSE Exception apresentado na hierarquia de exceções, na Figura 1, merece uma atenção especial. Estes são todos eventos recuperáveis, sendo que o grupo de exceções em Exception é checado pelo compilador, mas o grupo em RunTimeException não. Isto é, as exceções do primeiro grupo devem ser tratadas de alguma maneira, caso contrário o programa não será compilado; já o segundo grupo pode ou não ser tratado. Caso uma exceção não seja tratada de alguma forma, o programa, ou um evento do programa será encerrado com erro. Veja a seguir dois exemplos de exceções não tratadas e as mensagens de erro geradas.

```
public class Excecoes {
    public static void main(String[] args){
        int valor = 516 / 0;
        System.out.println("valor: "+valor);
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at excecoes.Excecoes.main(Excecoes.java:6)
```

```
public class Excecoes {
    public static void main(String[] args){
        FileReader f = new FileReader("meu-arquivo.txt");
    }
}
```

```
Unreported Exception FileNotFoundException; must be caught or declared to be thrown
```

1.1.Tratamento de Exceções Local

Vamos iniciar entendendo como tratar exceções em um programa Java de forma local, dentro do método no qual o erro/exceção ocorreu. Para tanto vamos precisar utilizar um conjunto de palavras-chaves: *try*, *catch* e eventualmente *finally*. A palavra-chave *try* é usada para definir um bloco de comandos que pode gerar um erro ou exceção. A palavra-chave *catch* é usada para definir qual tipo de exceção será capturada e como ela será tratada. Já a palavra-chave *finally* é utilizada para definir um bloco de instruções que devem ser executados independentemente da ocorrência ou não de exceções. Um exemplo de tratamento de exceções é apresentado no quadro abaixo.

Arquivo: Main.java

```
 1.  package excecoes;
 2.
 3.  public class Main {
 4.     static void alterValue(int[] array, int index, int value){
 5.        try{
 6.           array[index] = value;
 7.        } catch( IndexOutOfBoundsException exception ){
 8.           System.err.println("Exceção ao alterar valor!");
 9.           System.err.printf("indice: %d / ", index);
10.           System.err.printf("valor: %d \n", value);
11.           System.err.println("Erro: " + exception.toString());
12.        } finally{
13.           System.out.println("Saindo da função alterValue()");
14.        }
15.     }
16.
17.     public static void main(String[] args){
18.        int[] numbers = new int[10];
19.        alterValue(numbers,  5, 11); // ok, indice válido
20.        alterValue(numbers, 15, 21); // ops, indice inválido
21.     }
22.  }
```

O exemplo acima mostra um método que recebe um array, um índice e um valor, e altera o valor do array no índice especificado. Um problema que pode ocorrer é que o tamanho do vetor pode ser menor do que o índice passado como argumento para o método, gerando um acesso a uma

posição inválida da memória. Para tratar este erro foi adicionado um bloco *try catch*. Se um erro ocorrer na linha 6 ele será tratado por algum *catch* especificado abaixo. Note que pode haver mais de *catch* consecutivo, porém o erro será tratado apenas por um deles, o primeiro que for válido (de cima para baixo). No exemplo foi utilizado a classe de exceções IndexOutOfBoundsException, uma classe derivada específica para o problema de acesso fora dos limites de um array. Note que poderiam ter sido utilizadas as superclasses de exceções Throwable, ou Exception, garantindo que basicamente qualquer tipo de erro fosse tratado no bloco de código descrito. Entretanto, o uso das superclasses de forma isolada, impossibilita o tratamento específico de erros distintos.

OBS: A exceção mais genérica usualmente utilizada é **Exception**, pois Throwable inclui eventos de situações irrecuperáveis.

Você notou o uso da instrução `System.err.println` e `System.err.printf`? Por que foi utilizado `err` no lugar de `out`? Isso indica que é uma saída de erro. No console do netbeans, por exemplo, este tipo de saída é mostrado com uma fonte de cor vermelha. É interessante distinguir entre mensagens de saída ou erro.

É válido dizer ainda que o bloco *try* pode conter mais de um comando ou linha de código. Como mais de um tipo de erro pode ocorrer em um único comando, ou vários comandos do bloco *try*, é comum tratar cada erro ou grupo de erro específico de forma isolada. Para isto são especificados um ou mais blocos *catch*, um abaixo do outro. Entretanto, lembre-se de tratar sempre os erros mais específicos (classes derivadas) antes dos mais genéricos (superclasses), pois o primeiro bloco com tipo de exceção compatível será executado quando uma exceção ocorrer.

Exemplo Ilustrativo

```
 1.  try {
 2.      // executa tudo ou até a linha onde ocorrer a exceção
 3.  } catch ( TipoExcecao1 ex ) {
 4.      // executa se ocorrer uma exceção do tipo TipoExcecao1
 5.  } catch ( TipoExcecao2 ex ){
 6.      // executa se ocorrer uma exceção do tipo TipoExcecao2
 7.  } finally {
 8.      /* executa sempre, mesmo se houver um return nos blocos
 9.         acima. Utilizado usualmente para "limpeza": fechar
10.         arquivos, liberar recursos etc. */
11.  }
     // executa quase sempre. Não executa se uma exceção não for
     capturada ou se houver um return nos blocos acima.
```

1.2. Arremessando Exceções

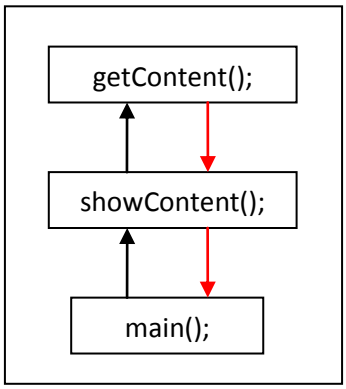
Quando uma exceção não é tratada por um bloco *try catch*, ela pode (exceções não checadas) ou precisa (exceções checadas) ser arremessada / lançada (*throw*) adiante de modo que outro método faça o seu devido tratamento. A sintaxe em Java que realiza isto é a palavra reservada *throws*, que deve ser inserida após a lista de parâmetros e antes da abertura do bloco da função. Após a palavra-chave *throws*, devem constar ainda os nomes das classes dos tipos de exceções que serão arremessadas adiante. O quadro abaixo mostra um exemplo no qual foi implementado um método estático *getContent()* que abre um arquivo especificado por parâmetro e retorna seu

conteúdo na forma de uma única String. Note que não foi utilizado um bloco *try catch* dentro deste método para tratar as possíveis exceções de FileNotFoundException e IOException, que podem ocorrer na criação de um objeto FileReader ou no método *read()*, respectivamente.

Arquivo: Excecoes.java

```
 1.  package excecoes;
 2.  import java.io.FileNotFoundException;
 3.  import java.io.FileReader;
 4.  import java.io.IOException;
 5.
 6.  public class Excecoes {
 7.
 8.      static String getContent(String dir) throws IOException {
 9.          FileReader f = new FileReader(dir);
10.          String content = "";
11.          while( f.ready() )
12.              content += (char) f.read();
13.          return content;
14.      }
15.
16.      static void showContent(String dir) throws IOException {
17.          String text = getContent(dir);
18.          System.out.println("Texto do arquivo:");
19.          System.out.println(text);
20.      }
21.
22.      public static void main(String[] args) {
23.          try{
24.              showContent("meu-arquivo.txt");
25.          } catch(FileNotFoundException ex){
26.              System.err.println("O arquivo não foi encontrado!");
27.              System.err.println(ex.getMessage());
28.          } catch(IOException ex){
29.              System.err.println("Exceção em oper. com arquivo");
30.              System.err.println(ex.getMessage());
31.          }
32.      }
33.  }
```

Polimorfismo: a exceção FileNotFoundException que pode ocorrer ao criar um objeto FileReader pode ser arremessada como uma IOException, pois é uma subclasse desta segunda.



Para efetivamente entender a prática de arremessar exceções é necessário compreender como funciona a pilha de execução de um programa. Aproveitando o exemplo apresentado no quadro acima, note que o programa irá começar no método *main()*, o qual faz uso do método estático *showContent()*, que por sua vez utiliza o método estático *getContent()*. Sempre que um método é invocado, ele é empilhado na pilha de execução do programa, e quando o método termina normalmente ou por execução, ele é removido da pilha de execução e retorna o fluxo de execução para o método imediatamente abaixo dele na pilha, que é exatamente o método que fez a sua chamada. Quando uma exceção é lançada no método *getContent()*, a execução do programa retorna ao método *showContent()*, na linha 17. Como a exceção gerada não é tratada por um bloco *try catch*, mas sim arremessada novamente, o método *showContent()* é finalizado e o fluxo de execução volta para a função *main()*, na linha 24. Neste trecho do código existe a especificação de blocos de

tratamento de exceções; logo, a exceção que foi arremessada de *getContent()* para *showContent()* e *showContent()* para *main()* será tratada em um dos blocos de *catch*.

1.4. Criando uma Exceção

Até o momento discutimos como tratar exceções que são geradas por erros lógicos ou pelos métodos de classes nativas do Java. Mas e se quisermos fazer uso deste mecanismo, mas gerando uma exceção própria e personalizada para algum método implementado? Isto pode ser feito fazendo uso novamente do conceito de arremessar exceções e da possibilidade de instanciar objetos de exceções. Neste sentido podemos optar por utilizar as classes de exceções já existentes do Java, ou criar a nossa própria classe de exceção.

Se estivermos modelando uma conta bancária, por exemplo, e quisermos proibir o saque de uma quantia maior do que o saldo existente, poderíamos gerar uma exceção de uma classe já existente que indica o uso de um parâmetro inválido e lançá-la caso esta situação ocorra. O código apresentado no quadro abaixo mostra uma possível implementação.

Arquivo: Conta.java

```
 1.  package excecoes;
 2.
 3.  public class Conta {
 4.      private int saldo;
         //...
16.      public void saque(int valor){
17.          if( valor > saldo )
18.              throw new IllegalArgumentException("Saque maior
19.                                                 do que o saldo");
20.          else
21.              saldo -= valor;
22.      }
23.  }
```

Arquivo: Main.java

```
     //...
13.  Conta cc = new Conta();
14.  cc.deposita(100);
15.  try{
16.      cc.saque( 500 );
17.  } catch(IllegalArgumentException ex){
18.      System.err.println( ex.getMessage() );
19.  }
     //...
```

Outra abordagem seria criar e utilizar uma classe de exceção personalizada. Para isto, deve-se criar uma classe comum e deve-se adicioná-la na hierarquia das classes de exceções através de herança, estendendo uma classe já existente da hierarquia. Para seguir com o exemplo, considere a criação de uma nova classe chamada InsufficientFund que estende RuntimeException. Com esta, poderíamos simplesmente trocar o nome das classes do exemplo acima. Perceba que a criação de uma nova classe de Exceção permite a definição de métodos e atributos que podem ser utilizados ao lançar ou capturar exceções. No exemplo ilustrado no quadro a seguir, entretanto, foi criado apenas um método construtor simples que recebe uma mensagem indicativa do erro.

Arquivo: InsufficientFund.java

```
1.  package excecoes;
2.  public class InsufficientFund extends RuntimeException {
3.      public InsufficientFund( String msg ){
4.          super( msg );
5.      }
6.  }
```

Arquivo: Conta.java

```
    //...
    throw new InsufficientFund("Saque maior do que o saldo");
    //...
```

1.5. Considerações Adicionais

Algumas considerações merecem ser destacadas:

- Um método sobrescrito não pode arremessar mais exceções do que os métodos originais;
- Evite arremessar exceções genéricas como as classes Throwable ou Exception;
- Como todo objeto de exceção herda de Throwable é importante conhecer dois métodos muito utilizados desta classe. São eles:
  - `String getMessage()` – retorna mensagem passada pelo construtor;
  - `void printStackTrace()` – mostra detalhes (*stack trace*) no console sobre a exceção;
- Lembre-se que exceções de RuntimeException e Error não são verificadas em tempo de compilação e portanto merecem atenção redobrada a fim de evitar a sua ocorrência sem tratamento. Já as exceções de Exception são verificadas em tempo de compilação sendo evidente a necessidade de tratar tal tipo de exceção no código fonte;
- O lançamento de exceções permite evitar a criação de um objeto inválido. Sem exceções, o máximo que se pode fazer é definir um valor padrão caso os valores repassados para o construtor sejam inválidos. Já com o uso de exceções pode-se interromper a criação do novo objeto gerando um objeto de erro/exceção.

2. GENERICS

Considere uma situação em que você deseja criar uma classe descritora que recebe um valor e uma descrição de um item qualquer, ou seja, qualquer objeto. A fim de atender as restrições impostas seria necessário utilizar o supertipo Object para armazenar qualquer tipo de objeto. Uma possível implementação é apresentada no quadro a seguir. O problema nesta implementação está no uso dos objetos desta classe, mas especificamente no método que retorna o objeto em questão. Este método é problemático, pois ele retorna um Object que é um supertipo. Esse tipo de retorno pode gerar dois problemas: (1) será sempre necessário fazer *downcasting* explícito para o tipo do objeto que foi armazenado e (2) pode ser difícil ou trabalhoso descobrir qual o tipo de objeto que foi armazenado para fazer *downcasting*; e caso um tipo seja inferido de forma incorreta isto poderá resultar em uma exceção no programa.

Arquivo: Descriptor.java

```
 1.  package generics;
 2.
 3.  public class Descriptor {
 4.      private Object object;
 5.      private String description;
 6.
 7.      public Descriptor(Object obj, String desc){
 8.          object = obj;
 9.          description = desc;
10.      }
11.      public Object getObject(){ return object; }
12.      public String getDescription(){ return description; }
13.      public void setDescription(String desc){
14.          description = desc;
15.      }
16.
17.      public static void main(String[] args){
18.          Descriptor descritor1 = new Descriptor(1, "Número um");
19.          Descriptor descritor2 = new Descriptor('A', "Letra A");
20.          int valor = (int) descritor1.getObject();
21.          char letra = (char) descritor2.getObject();
22.      }
23.  }
```

Para simplificar este tipo de situação felizmente existe o conceito de tipos genéricos. O nome deste conceito já oferece uma boa dica do seu propósito. Trata-se de uso de um tipo genérico que pode ser definido no momento da instanciação de um objeto da classe em questão. Java e outras linguagens de programação utilizam a notação de diamantes (< >) para representar tipos genéricos.

Para especificar que uma classe utiliza um tipo genérico deve-se adicionar a notação de diamantes logo após o nome da classe com uma letra ou palavra no interior, nomeando o tipo genérico a ser utilizado. É comum o uso das seguintes letras: E (representa *element*), K (representa *key*), N (representa *number*), T (representa *type*), V (representa *value*), S, U, V etc. (representam segundo, terceiro e quarto tipos). A letra ou nome utilizado para o tipo genérico deve ser utilizada como se fosse um tipo, ou uma classe. Caso seja necessário o uso de mais de um tipo genérico isso pode ser feito separando-os com vírgulas dentro do diamante. Para exemplificar, o código do arquivo Descriptor.java, mostrado acima, foi modificado a fim de usar Generics.

O uso de classes que utilizam tipos genéricos é ligeiramente diferente do normal. Ao instanciar um objeto de uma classe com tipos genéricos, pode-se determinar entre diamantes qual é o tipo composto a ser utilizado pela classe, substituindo o tipo genérico. Pode-se entender que todos os elementos genéricos são sobrescritos pelo tipo especificado. Desta forma, ao usar um método que retorna um tipo genérico, o compilador saberá que o tipo esperado é o tipo/classe especificado entre diamantes, não sendo necessário o uso de *downcasting* ou requerendo a descoberta da classe correta. Caso a instanciação do objeto seja feita de maneira ordinária, sem especificar um tipo composto, o supertipo Object será utilizado de maneira implícita. O quadro de código a seguir apresenta também o uso da nova classe Descriptor com tipo genérico. Note que atualmente o Java permite que, ao instanciar um objeto envolvendo Generics, o segundo diamante possa aparecer sem os tipos, inferindo-se que eles são os mesmos especificados no primeiro diamante.

Arquivo: Descriptor.java (atualização)

```
 1.  package generics;
 2.
 3.  public class Descriptor<T> {
 4.      private T object;
 5.      private String description;
 6.
 7.      public Descriptor(T obj, String desc){
 8.          object = obj;
 9.          description = desc;
10.      }
11.      public T getObject(){ return object; }
12.      public String getDescription(){ return description; }
13.      public void setDescription(String desc){
14.          description = desc;
15.      }
16.
17.      public static void main(String[] args){
18.          Descriptor<Integer> descritor1 =
19.                                  new Descriptor<>(1, "Número um");
20.          Descriptor<Character> descritor2 =
21.                                  new Descriptor<>('A', "Letra A");
22.          Descriptor d3 = new Descriptor("Ola", "Texto Ola");
23.          int valor = descritor1.getObject();
24.          char letra = descritor2.getObject();
25.      }
26.  }
```

2.1. Garantias sobre um Tipo Genérico

Em algumas situações é importante que tenhamos certas garantias sobre os tipos genéricos especificados. Um exemplo clássico é quando queremos comparar dois valores que são de tipos genéricos. O compilador do Java não vai permitir tal comparação, pois (a princípio) não sabe como deve realizar a comparação uma vez que inteiros são comparados de uma forma, Strings de outra, sem falar das classes criadas pelos próprios desenvolvedores. Para poder realizar uma comparação entre esses dois valores, é necessário garantir que tais valores sejam comparáveis. Ok, mas como garantir isto? Como garantir que uma classe composta tenha a definição de uma operação de comparação entre seus elementos? Se você pensou em métodos abstratos ou interfaces, acertou!

Podemos garantir que dois tipos (ou classes) sejam comparáveis, garantindo que elas implementem um método de comparação. Para efetivamente garantir isto, essas CLASSES devem implementar a interface Comparable que possui um método abstrato *compareTo()*. Assim, se uma classe implementa tal interface, podemos garantir que é possível comparar dois objetos deste tipo. Certo, mas como dizer isso a notação de Generics? Para isto usamos a palavra-chave *extends*. Caso exista mais de uma garantia a ser validada para um dado tipo genérico, é possível especificar todas as interfaces e classes abstratas após a palavra *extends* separadas pelo símbolo &.

O quadro a seguir mostra uma nova versão da classe Descriptor, garantindo que o tipo genérico T seja numérico (herda da classe Numeric) e comparável (implementa a interface Comparable). Como a interface Comparable será certamente implementada pela classe a substituir o

tipo genérico é possível afirmar que o método *compareTo()* estará implementado e assim, é possível comparar dois objetos instancias de Descriptor.

Arquivo: Descriptor.java (atualização)

```
 3.  public class Descriptor <T extends Number & Comparable> {
 4.      private T object;
 5.      //...
16.      public int compare(T item){
17.          if( item.intValue() < 0 ) return -1;
18.          return object.compareTo(item);
19.      }
         //...
36.  }
```

Com o uso de tais garantias o compilador verifica se o tipo utilizado realmente implementa as interfaces ou herda das classes abstratas especificadas. Considerando a nova atualização da classe Descriptor mostrada acima, teríamos que uma String não poderia ser utilizada para a instância de um objeto Descriptor, por exemplo. Isto porque a classe String implementa a interface Comparable, mas não herda da classe abstrada Number. A classe de inteiros (Integer), por outro lado, contempla as duas condições.

Exercícios

1) Crie uma representação de conta bancária que possui os seguintes atributos: nome do cliente, endereço, telefone, saldo, limite de crédito, saldoEmprestimo (utilizado para controlar o montante não pago de um empréstimo), limiteEmprestimo. A classe deve ter um construtor que recebe os dados do cliente (nome, endereço, telefone) e deve atribuir os valores aos demais atributos: saldo e saldo de empréstimo zerado, limite de crédito inicial de R$100 e limite de empréstimo de R$1000. Verifique no construtor se o nome possui pelo menos espaço em branco e se o telefone tem pelo menos oito dígitos. Em caso de inconformidade lance uma exceção de argumentos inválidos.

A classe conta bancária também deve ter alguns métodos: `saque(valor)`, `deposito(valor)`, `emprestimo(valor)`, `pagarEmprestimo(valor)`, `static transferência(conta, conta, valor)`. O método de saque deve verificar se o valor requisitado está disponível considerando o saldo e o limite de crédito da conta. O método de empréstimo deve verificar se o valor requisitado pode ser emprestado considerando o saldo de empréstimo e o limite de empréstimo. O método de transferência deve verificar se existe saldo suficiente na conta de onde o dinheiro está saindo. Para cada caso, se houver alguma inconsistência, lance uma exceção personalizada com um nome e mensagem apropriada. Note que em um empréstimo, o valor requerido entra efetivamente na conta do cliente, e este valor é acrescido de uma taxa de 20% no seu saldo de empréstimo (ex: emprestimo de R$500, saldo += 500 e saldoEmprestimo -= 500*1.2);

2) Implemente uma classe utilitária chamada Trio. Ela deve armazenar três valores de tipos possivelmente diferentes e especificados pelo programador. Deve conter um método construtor no qual são passados os três elementos, três métodos setters (*setFirst*, *setSecond*, *setThird*) para atribuir valores ao primeiro, segundo e terceiro atributo, e três métodos getters (*getFirst*, *getSecond*, *getThird*) para resgatar os valores do primeiro, segundo e terceiro atributo. Sobrescreva o método *toString()* de forma a mostrar um trio no seguinte formato: "Trio: (atributo1 , atributo2 , atributo3)"

Além disso, implemente também dois métodos estáticos – *min()* e *max()* – que recebem um vetor de elementos Trio e retornam o menor menor ou maior valor, respectivamente. Para comparar dois elementos Trio, a fim de achar o menor e maior valor, crie um novo método (público) *compareTo()*, que compara duas instancias de Trio. O método deve analisar qual instância possui o primeiro atributo menor/maior. Se o valor for igual, o método deve comparar da mesma forma o segundo atributo e, em caso de novo empate, analisar o terceiro atributo. Lembre-se de utilizar tipos genéricos.

Por fim, faça uso da classe criada e de seus métodos. Crie algumas instâncias de Trio's e utilize os *setters* e *getters*, e os métodos *min()* e *max()*.